CSE 544
Principles of Database Management Systems

Magdalena Balazinska

Winter 2009

Lecture 9 - Transactions: concurrency control

Where We Are

- Relational model
  - The relational model and other data models
  - Database design (real-world entities → relational schema)
- DBMS architecture
  - Overview
  - Storage and indexing
  - Query execution
  - Query optimization
- Next two lectures we will talk about **transactions**

References

- **Concurrency control and recovery.**

  Michael J. Franklin. The handbook of computer science and engineering. A. Tucker ed. 1997

- **Database management systems.**

  Ramakrishnan and Gehrke.

  Third Ed. **Chapters 16 and 17.**

Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

Motivating Example

```
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
```

```
SELECT sum(money)
FROM Budget
```

Would like to treat each group of instructions as a unit

Definition

- **A transaction** = one or more operations, single real-world transition

- Examples
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)

Transactions

- Major component of database systems
- Critical for most applications; arguably more so than SQL

- Turing awards to database researchers:
  - Charles Bachman 1973
  - Edgar Codd 1981 for inventing relational dbs
  - Jim Gray 1998 for inventing transactions

Transaction Example

```
START TRANSACTION
UPDATE Budget SET money = money - 100
WHERE pid = 1
UPDATE Budget SET money = money + 60
WHERE pid = 2
UPDATE Budget SET money = money + 40
WHERE pid = 3
COMMIT
```

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
  - Database returns to a state without any of the changes made by the transaction

Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when app program finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

ACID Properties

- Atomicity: Either all changes performed by transaction occur or none occurs
- Consistency: A transaction as a whole does not violate integrity constraints
- Isolation: Transactions appear to execute one after the other in sequence
- Durability: If a transaction commits, its changes will survive failures

- Q: Benefits & drawbacks of providing ACID transactions?

What Could Go Wrong?

- Why is it hard to provide ACID properties?

- Concurrent operations
  - Isolation problems
  - We saw one example earlier

- Failures can occur at any time
  - Atomicity and durability problems
  - Next lecture

- Transaction may need to abort

Different Types of Problems

```
Client 1: INSERT INTO SmallProduct(name, price)
          SELECT pname, price
          FROM Product
          WHERE price <= 0.99

          DELETE Product
          WHERE price <=0.99

Client 2: SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
```

What could go wrong ?

Inconsistent reads

Different Types of Problems

```
Client 1:
          UPDATE Product
          SET Price = Price – 1.99
          WHERE pname = ‘Gizmo’

Client 2:
          UPDATE Product
          SET Price = Price*0.5
          WHERE pname=‘Gizmo’
```

What could go wrong ?

Lost update

Different Types of Problems

```
Client 1:       UPDATE SET Account.amount = 1000000000
                WHERE Account.number = ‘my-account’
```

Aborted by system

```
Client 2:       SELECT Account.amount
                FROM Account
                WHERE Account.number = ‘my-account’
```

What could go wrong ?

Dirty reads

Types of Problems: Summary

- Concurrent execution problems
  - Write-read conflict: dirty read
    - A transaction reads a value written by another transaction that has not yet committed
  - Read-write conflict: unrepeatable read
    - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  - Write-write conflict: lost update
    - Two transactions update the value of the same object. The second one to write the value overwrite the first change
- Failure problems
  - DBMS can crash in the middle of a series of updates
  - Can leave the database in an inconsistent state

Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

Serializable Execution

- **Serializability**: interleaved execution has same effect as some serial execution

- Schedule of two transactions (Figure 1)

  $r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow$
  $\rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0$

- Serializable schedule: equiv. to serial schedule

  $r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow$
  $\rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1$

Implementation: Locking

- Can serve to enforce serializability
- Two types of locks: **Shared and Exclusive**
- Also need **two-phase locking (2PL)**
  - Rule: once transaction releases lock, cannot acquire any additional locks!
  - So two phases: growing then shrinking
- Actually, need **strict 2PL**
  - Release all locks when transaction commits or aborts

Deadlocks

- Two or more transactions are waiting for each other to complete
- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting
- **Deadlock detection**
  - Timeouts
  - Wait-for graph
    - This is what commercial systems use (they check graph periodically)

Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.

- Example:
  - T0: reads list of books in catalog
  - T1: inserts a new book into the catalog
  - T2: reads list of books in catalog
    - New book will appear!

- Can this occur?
- Depends on locking details (eg, granularity of locks)
- To avoid phantoms needs **predicate locking**

Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
  - Golden standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are only a few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often **default**)
  - Sometimes are hard to understand

Degrees of Isolation

- **Four levels of isolation**
  - All levels use **long-duration exclusive locks**
  - READ UNCOMMITTED: no read locks
  - READ COMMITTED: short duration read locks
  - REPEATABLE READ:
    - Long duration read locks on individual items
  - SERIALIZABLE:
    - All locks long duration and lock predicates

- **Trade-off: consistency vs concurrency**
- Commercial systems give choice of level

Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- Coarse grain locking (e.g., tables)
  - Many false conflicts
  - Less overhead in managing locks
- Alternative techniques
  - Hierarchical locking (and intentional locks) [commercial DBMSs]
  - Lock escalation

The Tree Protocol

- An alternative to 2PL, for tree structures
- E.g. B+ trees (the indexes of choice in databases)

- Because
  - Indexes are hot spots!
  - 2PL would lead to great lock contention

  - Also, unlike data, the index is not directly visible to transactions
  - So only need to guarantee that index returns correct values

The Tree Protocol

Rules:

- A lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)
- Cannot relock a node for which already released a lock
- “Crabbing”
  - First lock parent then lock child
  - Keep parent locked only if may need to update it
  - Release lock on parent if child is not full

- The tree protocol is NOT 2PL, yet ensures conflict-serializability !

Outline

- Transactions motivation, definition, properties
- Concurrency control and locking
- Optimistic concurrency control

Optimistic Concurrency Control

Validation-based technique

- Phase 1: Read
  - Transaction reads from database and writes to a private workspace
- Phase 2: Validate
  - At commit time, system performs validation
  - Validation checks if transaction could have conflicted with others
    - Each transaction gets a timestamp
    - Check if timestamp order is equivalent to a serial order
  - If there is a potential conflict: abort
- Phase 3: Write
  - If no conflict, transaction changes are copied into database

Optimistic Concurrency Control

Timestamp-based technique

- Each object, O, has read and write timestamps: RTS(O) and WTS(O)
- Each transaction, T, has a timestamp TS(T)

- Transaction wants to read object O
  - If TS(T) < WTS(O) abort
  - Else read and update RTS(O) to larger of TS(T) or RTS(O)

- Transaction wants to write object O
  - If TS(T) < RTS(O) abort
  - If TS(T) < WTS(O) ignore my write and continue (Thomas Write Rule)
  - Otherwise, write O and update WTS(O) to TS(T)

Optimistic Concurrency Control

Multiversion-based technique

- Object timestamps: RTS(O) & WTS(O); transaction timestamps TS(T)
- Transaction can read most recent version that precedes TS(T)
  - When reading object, update RTS(O) to larger of TS(T) or RTS(O)
- Transaction wants to write object O
  - If TS(T) < RTS(O) abort
  - Otherwise, create a new version of O with WTS(O) = TS(T)
- Common variant (used in commercial systems)
  - To write object O only check for conflicting writes not reads
  - Use locks for writes to avoid aborting in case conflicting transaction aborts

Commercial Systems

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL:
  - Multiversion concurrency control
- Oracle
  - Multiversion concurrency control