More Transactions and Java Concurrency

CS186 Week 10

Overview

- Transactions
  - Phantom Problem
  - Multigranular Locking
  - Optimistic CC
  - Snapshot Isolation
- Java Concurrency

Phantom Problem

T1	R1			R1	R2
T2		W2	COMMIT		

What could go wrong with the following plan under Strict 2pl?

Phantom Problem

T1	R1			R1	R2
T2		W2	COMMIT		

R1: SELECT COUNT(*) FROM sailors WHERE rating=1;

W2: INSERT INTO sailors VALUES('joe', 1);

R1: SELECT COUNT(*) FROM sailors WHERE rating=1;

Phantom Problem

T1	R1			R1	R2
T2		W2	COMMIT		

R1: SELECT COUNT(*) FROM sailors WHERE rating=1;

W2: INSERT INTO sailors VALUES('joe', 1);

R1: SELECT COUNT(*) FROM sailors WHERE rating=1;

No way for this to be serializable: First time reading T1 sees only original tuples. Cannot possibly know beforehand that W2 in T2 will result in an extra tuple for rating=1.

Multiple-Granularity Locks

- Shared (S)
- Exclusive (X)
- IS
  - Intent to get a shared lock at a finer level
  - When is this used?
    - Index scan, reading only specific pages of the table
- IX
  - Intent to get exclusive lock at a finer level
  - When is this used?
    - Inserting tuples
- SIX
  - Shared lock on every child (e.g. if SIX lock on Sailors table, then all pages and tuples of Sailors get the S lock.)
  - Descendants can have X or IX status
  - When is this used?
    - Scanning and updating a table

Multiple-Granularity Locks

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

Ex 1: S**ELECT * FROM ratings;** gets an S lock on the entire ratings table.

INSERT INTO ratings('joe', 10); can only get IX lock (the intention to lock) after the first query finishes before it can get IX lock to even consider writing to any page in the ratings table.

Ex 2: **SELECT * FROM ratings where rating=5;** Get an IS lock on the ratings table (since we don't need to look at the entire table) and we get an S lock on the pages of the ratings table that have tuples with rating = 5.

INSERT INTO ratings('joe', 10); can now get IX lock on ratings and now can get X lock a page of ratings that is not S locked by the first query and insert a tuple onto that page.

Snapshot Isolation

- When a transaction starts, it’s given a snapshot of the DB.
- When you commit items, those items must not have changed since your snapshot.
  - If T1 and T2 start with the same snapshot and both modify the same tuples, then only one transaction can commit.
- Benefits:
  - Reads never blocked, and they don’t block writes.
  - No dirty read, inconsistent read, or phantoms.

Optimistic CC

- No locks. Write to private copies of data and merge at the end, hoping for no conflicts.
- Three phases:
  - READ – Read from DB, make private copies of changes.
  - VALIDATE – Check for conflicts.
  - WRITE – Merge all private changes into DB.

Questions?

What is concurrency?

- Concurrency vs. Parallelism
  - Concurrency: Two tasks run in overlapping/non-overlapping time periods
  - Parallelism: Two tasks literally at the same time
  - Parallel programs are a subset of concurrent programs.
- Why is DBMS multithreaded?
  - Queries can be run concurrently
  - e.g. Scanning through two tables is embarrassingly parallel
  - Not all transactions have as many conflicts as the ones we show you :)

Java Concurrency - Fields

- volatile
  - If multiple threads modify the same object, the object's non-volatile fields may be cached locally by a thread. Declaring fields to be volatile means that if thread 1 modifies a volatile field in an object, thread 2's local copy of that volatile field will be updated as well.
- static vs static volatile
  - A static field means there's only one copy right?
    - WRONG! Threads can also keep local cached copies of static fields. If you want to use a static field over multiple threads you must declare it to be volatile.
- final
  - Final fields cannot change, so can safely be accessed. Local cached copies of final fields are always up to date, since they cannot change.

Java Concurrency - Code

- synchronized
  - You can have synchronized blocks or synchronized methods (a critical region)
  - If one thread is running a synchronized method for object A (e.g. A. increment()), all other threads cannot execute another synchronized method on object A.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Java Concurrency - Code

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

```
public class SynchronizedCounter {
    private int c = 0;
    public void increment() {
        synchronized(this) {c++;}
    }
    public void decrement() {
        synchronized(c) { c--;}
    }
    public synchronized int value() {
        return c;
    }
}
```

Concurrent data structures

- java.util.concurrent.atomic (AtomicBoolean, AtomicInteger, AtomicIntegerArray) - reading, writing, compareAndSet, getAndAdd are all atomic operations. Atomic operations must be serial.
- BlockingQueue, ConcurrentHashMap
  - These help avoid Memory Consistency Errors where threads may not have a consistent view of the same data
- Locks
  - Essentially a lock