

CS290f - Lecture 11

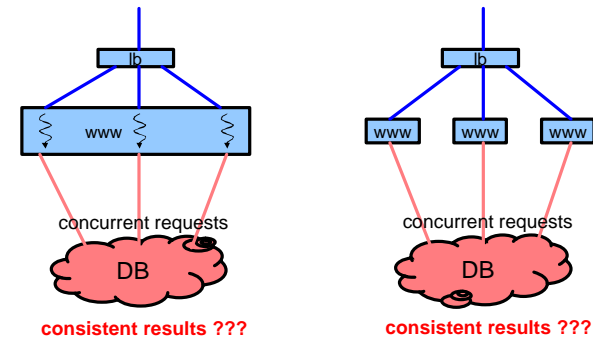
Database concurrency control

Scalable Internet Services, Fall 2006

Thorsten von Eicken
Department of Computer Science
University of California at Santa Barbara

Some database material from J. Hellerstein @UCB

The problem



Aside: why do we need concurrent database requests?

Database transactions

■ Background

- Concept that allow a system to guarantee certain semantic properties.
- Rigorously defined guarantees => can build correct systems using them

■ History

- Old hierarchical and network DBs had pretty clever systems for handling reliability
 - ◆ BUT -- no formalisms to describe the semantics clearly
 - ◆ Hence few lessons to transfer to other systems.
- IBM System R's RSS team, led by Jim Gray, codified the formal notion of *transactions* and *serializability* (mid '70s)
- System R delivered a working (though inefficient) implementation. Led to 1998 Turing Award for Gray.

ACID properties

■ A transaction should enjoy the following guarantees:

- **Atomicity**
 - ◆ the "all or nothing" property
 - ◆ programmer needn't worry about partial states persisting
- **Consistency**
 - ◆ the database should start out "consistent", and at the end of transaction remain "consistent"
 - ◆ definition of "consistent" is given by integrity constraints
- **Isolation**
 - ◆ a transaction should not see the effects of other uncommitted transactions
- **Durability**
 - ◆ once committed, the transactions effects should not disappear
 - ◆ (though they may be overwritten by subsequent committed transactions).

ACID properties cont.

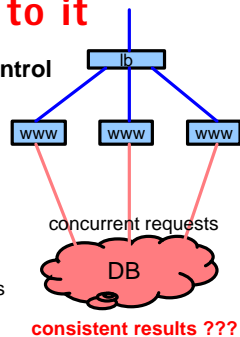
- **Atomicity, Consistency, Isolation, Durability**
- **ACID is a mnemonic**
 - not a perfect factoring of the issues
 - there is overlap of concerns among the four.
- **Implementation**
 - A and D are guaranteed by *recovery* (usually implemented via logging)
 - C and I are guaranteed by *concurrency control* (usually implemented via locking)
- **No help with side-effects**
 - actions that are visible outside the "system"
 - print to screen, send a web page, output money, communicate with web service, ...

The simple answer

- **Use transaction in Rails**
 - ```
transaction do
 david.withdrawal(100)
 mary.deposit(100)
end
```
  - Probably need `begin..rescue..end` wrapper around `transaction`

## ... but there's more to it

- **Use database for concurrency control**
  - Consistency
  - Isolation
- **Understand how databases implement these properties**
  - Better understand guarantees
  - Understand limitations
  - Understand performance characteristics



## Concurrency Control

- **A serial execution of transactions is safe but slow**
  - Essentially by definition
  - Try to find schedules equivalent to serial execution
- **Notion of "conflicting actions"**
  - Two or more actions are said to be in conflict if:
    - ◆ The actions belong to different transactions
    - ◆ At least one of the actions is a write operation
    - ◆ The actions access the same object (read or write)
  - Example of conflicting actions:
    - ◆ T1:R(X), T2:W(X), T3:W(X)
  - While the following sets of actions are not:
    - ◆ T1:R(X), T2:R(X), T3:R(X)
    - ◆ T1:R(X), T2:W(Y), T3:R(X)
- **Conflict => can't blindly execute in parallel**

## Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule **S** is **conflict serializable** if:
  - S is conflict equivalent to some serial schedule

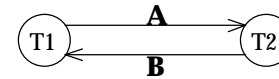
## Example

- A schedule that is not conflict serializable:

|     |                        |            |
|-----|------------------------|------------|
| T1: | R(A), W(A),            | R(B), W(B) |
| T2: | R(A), W(A), R(B), W(B) |            |

- It's **dependency graph**

- One node per Xact
- Edge from  $T_i$  to  $T_j$  if
  - ♦ an operation of  $T_i$  conflicts with an operation of  $T_j$  and
  - ♦  $T_i$ 's operation appears earlier in the schedule than the conflicting operation of  $T_j$



- **Theorem**

- A schedule is conflict serializable if and only if its dependency graph is acyclic

## Dependency Graph

- **Definition**
  - One node per Xact
  - Edge from  $T_i$  to  $T_j$  if
    - ♦ an operation of  $T_i$  conflicts with an operation of  $T_j$  and
    - ♦  $T_i$ 's operation appears earlier in the schedule than the conflicting operation of  $T_j$
- **Theorem**
  - A schedule is conflict serializable if and only if its dependency graph is acyclic

## Two-Phase Locking (2PL)

- 2PL allows only conflict serializable schedules

- **Two-Phase Locking Protocol**

- Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
  - ♦ If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object
- A transaction cannot request additional locks once it releases any locks
- Two phases: acquire locks, release locks

- **Issue: can result in "cascading aborts"**

- T1: R(A) W(A) unlock(A) ..... abort
- T2: R(A) ..... abort

## Strict Two-Phase Locking

- 2PL allows only conflict serializable schedules
- Two-Phase Locking Protocol
  - Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
    - ◆ If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object
  - All locks held by a transaction are released when the transaction completes

## Deadlocks

- Cycle of transactions waiting for locks to be released by each other
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection ← most common
- Mysql/Rails:
  - Mysql::Error: Lock wait timeout exceeded; try restarting transaction:
  - UPDATE ec2\_ssh\_keys SET `created\_at` = '2006-11-02 07:05:18', `aws\_material` = NULL, `aws\_fingerprint` = 'b6:88:a4:03:c7:42:8e:30:c4:a8:c2:a2:28:1a:b2:5c:3b:23:45:f3', `deleted\_at` = NULL, `aws\_key\_name` = 'ami-key', `account\_id` = 2, `updated\_at` = '2006-11-03 21:07:42' WHERE id = 1 - (ActiveRecord::StatementInvalid)

## Phantom rows

- Consider:

|                                    |                                     |
|------------------------------------|-------------------------------------|
| ● <u>THREAD A</u>                  | <u>THREAD B</u>                     |
| ● select * from tab where x > 100; |                                     |
| ●                                  | insert tab values (... x = 200 ...) |
| ● for each result row              |                                     |
| ● update ....                      |                                     |
- Problem
  - How do you lock something that doesn't exist?
  - Lock table, lock index, lock predicate, next-key locking

## Next-row locking in InnoDB

- Lock "gap" between keys in index
  - Locks set on an index key also locks "gap" before that key
    - ◆ (Can also lock "gap" after last key)
- Example
  - select \* from tab where x > 100 for update;
  - Assume index on x
  - InnoDB scans index and locks all keys > 100
  - This prevents phantom rows from appearing

## Concurrency control options

- **Reminder: for Consistency & Isolation guarantees**
- **Locking**
  - 2PL or Strict 2PL
  - Lock granularity
    - ◆ Row-level locks
    - ◆ Page-level locks
    - ◆ Table-level locks
- **Multi-version concurrency control**
  - Transactions see snapshot of database at time they started
    - ◆ Never wait on locks!
  - Implemented using copy on update (hence multiple versions)
  - Transactions may abort due to serialization failure...

17

## Transactions vs. locks

- **Transactions guarantee properties**
  - May use locks internally to implement guarantees
  - Do not provide control over lock acquisition
  - May abort due to “concurrency problems”
- **Locks provide mechanism**
  - Allow fine control over mechanisms used to “deal with” application concurrency
  - Leave a number of problems up to the programmer
    - ◆ E.g. deadlock

18

## Transaction isolation “levels”

- **Strict isolation can be expensive (slow)**
- **SQL Transaction Isolation Levels**

| Isolation Level  | Dirty Read   | Nonrepeatable Read | Phantom Read |
|------------------|--------------|--------------------|--------------|
| Read uncommitted | Possible     | Possible           | Possible     |
| Read committed   | Not possible | Possible           | Possible     |
| Repeatable read  | Not possible | Not possible       | Possible     |
| Serializable     | Not possible | Not possible       | Not possible |

- Dirty read: read value written by uncommitted transaction
- Non-repeatable read: another transaction writes after this transaction reads
- Phantom read: a new row appears due to another transaction
- **Levels available:**
  - PostgreSQL: read committed (default) and serializable
  - MySQL/InnoDB: all four, repeatable read is default

19

## Consistent vs. locking read

- **Note:**
  - Assume default repeatable read isolation level
  - Select results in consistent read using MVCC (and no locks)
  - Select for update results in locking read
- **Consistent read example**

time

**User A**

```
SET AUTOCOMMIT=0;
SELECT * FROM t;
empty set
COMMIT;
```

**User B**

```
SET AUTOCOMMIT=0;
INSERT INTO t VALUES (1, 2);
COMMIT;
```

```

| 1 | 2 |

1 row in set
```
- **If A had used locking reads, then B would have blocked on excl lock**

20

## Summary

### ■ Concurrency control is tricky

- Whether database or non-database
- Whether database tables and rows, or object collections and objects
- Whether multiple threads, processes, or machines

### ■ Compare transactions to object locks

- Transactions provide more guarantees than simple locks
- (One can also implement transaction-like concurrency control on in-memory objects)

### ■ The issue is deep

- Significant correctness impact
- Significant performance impact
- Lots of room for tradeoff
  - ◆ it all depends on the semantics of the app and the risk factor