CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

CONCURRENCY CONTROL

Lecture E

OPTIMISTIC CONCURRENCY CONTROL

Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Increases transactions per second
- For non-transaction systems, increases operations per second and lowers latency
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are *expected to be* rare
  - But still need to ensure conflicts are caught!

First-cut Approach

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from *those* transactions also now need to be aborted

    ☹ *Cascading aborts*

Second Approach: Timestamp Ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order
- Ensure that for a transaction T, both are true:
  1. T's write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T's read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
  - Can we do better?

Third Approach: Multi-version Concurrency Control

- For each object
  - A per-transaction version of the object is maintained
    - Marked as *tentative* versions
  - And a committed version
- Each tentative version has a timestamp
  - Some systems maintain both a read timestamp and a write timestamp
- On a read or write, find the “correct” tentative version to read or write from
  - “Correct” based on transaction id, and tries to make transactions only read from “immediately previous” transactions

Eventual Consistency …

- … that you've seen in key-value stores …
- … is a form of optimistic concurrency control
  - In Cassandra key-value store
  - In DynamoDB key-value store
  - In Riak key-value store
- But since non-transaction systems, the optimistic approach looks different

Eventual Consistency in Cassandra and DynamoDB

- Only one version of each data item (key-value pair)
- Last-write-wins (LWW)
  - Timestamp, typically based on *physical time*, used to determine whether to overwrite

```
if(new write's timestamp > current object's timestamp)
    overwrite;
else
    do nothing;
```

- With unsynchronized clocks
  - If two writes are close by in time, older write might have a newer timestamp, and might win

Eventual Consistency in Riak Key-value Store

- Uses vector clocks! (Should sound familiar to you!)
- Implements causal ordering
- Uses vector clocks to detect whether
  1. New write is strictly newer than current value, or
  2. If new write conflicts with existing value
- In case (2), a *sibling value* is created
  - Resolvable by user, or automatically by application (but not by Riak)
- To prevent vector clocks from getting too many entries
  - Size-based pruning
- To prevent vector clocks from having entries updated a long time ago
  - Time-based pruning

Summary

- RPCs and RMIs
- Transactions
- Serial Equivalence
  - Detecting it via conflicting operations
- Pessimistic Concurrency Control: locking
- Optimistic Concurrency Control