Weaker Consistency Models

Alex Bell

Brown University

ambell@gmail.com

April 30, 2012
Werner Vogels, Eventually Consistent, Communications of the ACM, 2009

- CTO, Amazon
- Blog: All Things Distributed
- Blog post, December 2007
Motivation

- Requirements for cloud computing:
  - Security
  - Scalability
  - Availability
  - Performance
  - Cost-effectiveness
  - *All while serving millions of users around the world!*

- Common solution: replication
  - Tradeoff: high availability and data consistency
Historical Context

- **1970s**: distribution transparency
  Better to fail whole system than break transparency

- **1990s**: internet → focus on availability
  - Eric Brewer’s CAP theorem: data consistency, system availability, tolerance to network partition
  - Problem: large-scale systems need to be network tolerant! Must choose from consistency or availability...
  - Solution: *relax* consistency guarantees
Consistency to a Client
Strong Consistency

After the update, any subsequent access will return the updated value.

credit: Wilfred Springer
Weak Consistency

The system does not guarantee that at any given point in the future subsequent access will return the updated value.

credit: Wilfred Springer
Eventual Consistency

If no updates are made to the object, eventually all accesses will return the last updated value.

credit: Wilfred Springer
Inconsistency Window until Eventual Consistency

- Can calculate based on metrics like communication delays, load on system, etc.
- If reading from asynchronous replica, inconsistency window = length of log shipment
Causal Consistency

Subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write.

credit: Wilfred Springer
Read-your-writes Consistency

Process A, after updating a data item always access the updated value and never sees an older value.
Session Consistency

Within the "session", the system guarantees read-your-writes consistency.

credit: Wilfred Springer
Monotonic Read Consistency

If a process has seen a particular value for the object, any subsequent access will never return any previous values.

credit: Wilfred Springer
Monotonic Write Consistency

The system guarantees to serialize the writes by the same process:

- **A**
  - 0: value = "foo"
  - 1: value = "bar"
  - 2: value = "last"

- **B**
- **C**

credit: Wilfred Springer
Consistency to a Server

Weaker Consistency Models
Model

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Total replicas</td>
</tr>
<tr>
<td>$W$</td>
<td># must acknowledge update before it commits</td>
</tr>
<tr>
<td>$R$</td>
<td># must poll when data object is read</td>
</tr>
</tbody>
</table>

$W + R > N$

$\rightarrow \exists$ a replica in read set that was in write set
$\rightarrow$ always retrieve latest value from at least one replica
$\rightarrow$ Strong Consistency

Ready for quiz / discussion?
Common Recipes

Try to use some imagination. There may be more than one answer. Also, extra points to whoever spots the error.

<table>
<thead>
<tr>
<th>N</th>
<th>W</th>
<th>R</th>
<th>What?</th>
<th>Consistent?</th>
<th>Tolerant?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Synchronous Replication</td>
<td>Yes</td>
<td>Depends</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1*</td>
<td>Asynchronous Replication</td>
<td>No</td>
<td>Depends</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>1</td>
<td>Optimized Reads</td>
<td>Yes</td>
<td>Write Outages</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>N</td>
<td>Optimized Writes</td>
<td>Yes</td>
<td>Read Outages</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>1</td>
<td>Lazy Updates (eventual)</td>
<td>No</td>
<td>Very</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Standard Fault Tolerance</td>
<td>Yes</td>
<td>1 failure</td>
</tr>
</tbody>
</table>

Any case in which $W + R \leq N$:

Weak/Eventual Consistency

---

*Vogel’s example: write to one specific node, read from either master or slave.
The Problem
Consistency to a Client
Consistency to a Server (& Quiz)
Questions
Megastore: Providing Scalable, Highly Available Storage for Interactive Services


Google Inc.

CIDR 2011, Jan. 12 2011
Goals

credit: Google
Goals

- Scalable
- Eventual Consistency
- Bigtable
- Clustering
- Wide-Area Replication
- Tx Consistent

credit: Google
Entity Groups

The smallest unit of data is an entity. Each entity has a set of properties. **Within an entity group, operations are transactional; weak (async) consistency for cross-group transactions.**

<table>
<thead>
<tr>
<th>Application</th>
<th>Entity Groups</th>
<th>Cross-EG Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td>Each account</td>
<td>None *</td>
</tr>
<tr>
<td>Blogs</td>
<td>Accounts, Blogs</td>
<td>Notifications, global indexes</td>
</tr>
<tr>
<td>Maps</td>
<td>Patches of Earth</td>
<td>Operations spanning large areas</td>
</tr>
</tbody>
</table>

* In this model, all that users do is send/receive mail; external mail handlers communicate between accounts.
Distributed Architecture

- Datacenters
- Entity Groups partition the datastore
- Each entity group is synchronously replicated across datacenters
- ACID semantics within an entity group
- Looser consistency across entity groups
- Bigtables in each Datacenter

credit: Google
Indexes

- **Local:** find data within E.G.
  - One for each E.G.
  - Updated synchronously

- **Global:**
  - Spans E.G.’s
  - Updated asynchronously (not guaranteed current)
Operations Across Entity Groups

Most transactions are within a single entity group.

Cross-entity group transactions supported via Two-Phase Commit.

Asynchronous communication between entity groups supported by Queues.

Global indexes span entity groups but have weaker consistency.

Entities (units of data)

Local Index

Entity Group 1

Global Indexes

Entity Group 2

credit: Google
Example Schema for an E.G.

```sql
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
    required int64 user_id;
    required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
    required int64 user_id;
    required int32 photo_id;
    required int64 time;
    required string full_url;
    optional string thumbnail_url;
    repeated string tag;
} PRIMARY KEY(user_id, photo_id), IN TABLE User,
    ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime ON Photo(user_id, time);
CREATE GLOBAL INDEX PhotosByTag ON Photo(tag) STORING (thumbnail_url);
```

credit: Google
Multi-Version Concurrency Control (aka timestamps)

Types of reads:

1. **Current**: previous commits applied; read at timestamp of last committed txn

2. **Snapshot**: read at timestamp of last known fully applied txn, even if some commits not yet applied

3. **Inconsistent**: just read latest values (low latency)
General Transaction Lifecycle

1. **Read:** Get timestamp, log position of last committed txn
2. **Application logic:** Read from Bigtable, gather writes into log entry
3. **Commit:** Use modified Paxos to achieve consensus for appending that entry to log
4. **Apply:** Write mutations to entities/indexes in Bigtable
5. **Clean up:** Delete any temporary data created above
Modified Paxos

- Fast reads (usually local)
  - Usually local: each replica’s Coordinator keeps track of whether it is up to date and can process local reads
  - Write algorithm maintains that state: if write fails on a replica, invalidates Coordinator

- Fast writes (single trip)
  - Leader replica chosen to arbitrate each log position
  - Leader piggybacks prepare phase of upcoming consensus onto accept phase of current consensus
Reads

Timeline of read algorithm

Client → Coordinator A → Replica A → Replica B → Replica C

Check Coordinator → Find Pos → Optional Majority Read

Catchup → Get Logs → Apply Logs

Validate → Query Data

credit: Google
Weaker Consistency Models
Performance

- 100+ app’s
- Highly available
- Median read latency: 70 milliseconds
- Median write latency: 350 milliseconds

Figure 9: Distribution of Availability
Figure 10: Distribution of Average Latencies
Motivation
Architecture
Transactional Algorithms
Performance
Questions

Weaker Consistency Models
Life Beyond Distributed Transactions
An Apostate’s Opinion
By Pat Helland
Amazon.Com
Jan 8th, 2007

Apostate: noun
“One who renounces a previously held belief.”

Today’s Goal:
Offer hopefully insightful opinions about scaleable apps.

credit: Pat Helland
Grown-Ups Don’t Use Distributed Transactions
- The apps using distributed transactions become too fragile…
- Let’s just consider local transactions.
  → Multiple Disjoint Scopes of Serializability

Want Almost-Infinite Scaling
- More of everything… Year by year, bigger and bigger
- If it fits on your machines, multiply by 10, if that fits, multiply by 1000…
- Strive to scale almost linearly (N log N for some big log).

Want Scale-Agnostic Applications
- Two layers to the application: scale-agnostic and scale-aware
- Consider Scale-Agnostic API

credit: Pat Helland
<table>
<thead>
<tr>
<th>Vocabulary and Assertions for Discussing Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Almost-Infinite Scaling</strong></td>
</tr>
<tr>
<td><strong>Scale-Agnostic App</strong></td>
</tr>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>Activity</strong></td>
</tr>
<tr>
<td><strong>Alternate Indices Aren’t Transactionally Consistent</strong></td>
</tr>
<tr>
<td><strong>Entities Cooperate Using Fine-Grained Two-Party Workflow</strong></td>
</tr>
</tbody>
</table>

**Credit:** Pat Helland

**Positions**
- Vogels: Eventually Consistent
- Google: Megastore
- Helland: Life Beyond Distributed Transactions

**Definitions**

**Questions**

**Distributed Operations**
Rethinking Cross-Entity Operations

- Entities are partitioned horizontally, never vertically
- Entities never share transaction scopes
  - *Scale-agnosticism* and *almost infinite scaling*
  - Plan for almost infinite repartitioning
- Regardless of physical location, entities connected by asynchronous messages
  - Per-partner messages are easier to scale
- What if messages are repeated? ie, network faults during communication round-trip
  - If idempotent (read), just resend response
  - Ensure mutation is not re-applied
- Like with Megastore, never assume cross-entity indexes are in sync
Uncertainty During Cross-Entity Operations?

- Traditionally, masked by DBMS
- Why not the app’s responsibility? This better models real world
  - Customer puts item in online shopping cart to reserve it
  - When checking inventory, manager must decide how to interpret transactions in limbo
- Application would confirm or cancel operation