Final Review + Systems Potpourri
Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 12th @11:59pm

Final Exam: Sunday Dec 16th @ 8:30am
FINAL EXAM

Who: You
What: http://cmudb.io/f18-final
When: Sunday Dec 16th @ 8:30am
Where: GHC 4401
Why: https://youtu.be/6yOH_FjeSAQ
What to bring:
→ CMU ID
→ Calculator
→ Two pages of handwritten notes (double-sided)

Optional:
→ Spare change of clothes

What not to bring:
→ Your roommate
Your feedback is strongly needed:
→ [https://cmu.smartevals.com](https://cmu.smartevals.com)

Things that we want feedback on:
→ Homework Assignments
→ Projects
→ Reading Materials
→ Lectures
OFFICE HOURS

Andy:
→ Friday Dec. 14th @ 3:00pm-4:00pm
STUFF BEFORE MID-TERM

SQL
Buffer Pool Management
Hash Tables
B+Trees
Storage Models
PARALLEL EXECUTION

Inter-Query Parallelism
Intra-Query Parallelism
Inter-Operator Parallelism
Intra-Operator Parallelism
EMBEDDED LOGIC

User-defined Functions
Stored Procedures

Focus on advantages vs. disadvantages
ACID
Conflict Serializability:
→ How to check?
→ How to ensure?
View Serializability
Recoverable Schedules
Isolation Levels / Anomalies
TRANSACTIONS

Two-Phase Locking
→ Strict vs. Non-Strict
→ Deadlock Detection & Prevention

Multiple Granularity Locking
→ Intention Locks
TRANS ACTIONS

Timestamp Ordering Concurrency Control
→ Thomas Write Rule

Optimistic Concurrency Control
→ Read Phase
→ Validation Phase
→ Write Phase

Multi-Version Concurrency Control
→ Version Storage / Ordering
→ Garbage Collection
CRASH RECOVERY

Buffer Pool Policies:
→ STEAL vs. NO-STEAL
→ FORCE vs. NO-FORCE

Write-Ahead Logging

Logging Schemes

Checkpoints

ARIES Recovery
→ Log Sequence Numbers
→ CLRs
DISTRIBUTED DATABASES

System Architectures
Replication
Partitioning Schemes
Two-Phase Commit
<table>
<thead>
<tr>
<th>Year</th>
<th>Software</th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>MongoDB</td>
<td>32</td>
<td>33</td>
<td>15</td>
<td>CockroachDB</td>
</tr>
<tr>
<td></td>
<td>Google Spanner/F1</td>
<td>22</td>
<td>22</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LinkedIn Espresso</td>
<td>16</td>
<td>19</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Apache Cassandra</td>
<td>16</td>
<td>17</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Facebook Scuba</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Redis</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>VoltDB</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Redis</td>
<td>10</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertica</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cloudera Impala</td>
<td>5</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cloudera Impala</td>
<td>5</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>LinkedIn Espresso</td>
<td>5</td>
<td>11</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LinkedIn Espresso</td>
<td>5</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Peloton</td>
<td>7</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Peloton</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>CockroachDB</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Google Spanner/F1</td>
<td>22</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amazon Aurora</td>
<td>18</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Redis</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LinkedIn Espresso</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CockroachDB</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Apache Cassandra</td>
<td>17</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElasticSearch</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Apache Cassandra</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Facebook Scuba</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MySQL</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table above lists various database software and their rankings from 2015 to 2018, with the highest ranked software in bold.
COCKROACHDB

Started in 2015 by ex-Google employees.
Open-source (Apache Licensed)
Decentralized shared-nothing architecture.
Log-structured on-disk storage (RocksDB)
Concurrency Control:
→ MVCC + OCC
→ Serializable isolation only
DISTRIBUTED ARCHITECTURE

Multi-layer architecture on top of a replicated key-value store.
→ All tables and indexes are stored in a giant sorted map in the k/v store.

Uses RocksDB as the storage manager at each node.

Raft protocol (variant of Paxos) for replication and consensus.
CONCURRENCY CONTROL

DBMS uses **hybrid clocks** (physical + logical) to order transactions globally.
→ Synchronized wall clock with local counter.

Txns stage writes as "intents" and then checks for conflicts on commit.

All meta-data about txns state resides in the key-value store.
COCKROACHDB OVERVIEW

Global Database Keyspace (Logical)

<table>
<thead>
<tr>
<th>System</th>
<th>Table1</th>
<th>Index1</th>
<th>Table2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key→Location</td>
<td>Key→Data</td>
<td>Key→Data</td>
<td>Key→Data</td>
</tr>
</tbody>
</table>

Node 1

Node 2

Node 3
COCKROACHDB OVERVIEW

Global Database Keyspace (Logical)

- **System**: Key→Location
- **Table1**: Key→Data
- **Index1**: Key→Data
- **Table2**: Key→Data

Application

- **Node 1**: Leader
- **Node 2**: 
- **Node 3**: 

Key/Value API
COCKROACCHDB OVERVIEW

Global Database Keyspace (Logical)

- System (Key→Location)
- Table1 (Key→Data)
- Index1 (Key→Data)
- Table2 (Key→Data)

Application

Node 1 (Leader)
Node 2
Node 3

Key/Value API

Raft
Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Chemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tesla Chandra, Andrew Fikes, Robert E. Gruber

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size, with properties of data access through commodity servers. Many projects at Google store data in Bigtable, including with indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from hulking bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple 10 principles model by which Bigtable, which grew from dynamic control over data layout and format, and always aims to improve the locality properties of the data represented in the underlying storage. Data is stored using row and column names for arbitrary strings. Bigtable also stores data in unstructured arrays, although clients often simplify various forms of structured and semi-structured data into these strings. Clients can control the location of their data through careful choices in their scheme. Finally, Schema change patterns are driven dynamically, and clients do not need to store data out of memory or from disk.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to scale to petabytes of data and thousands of machines. Bigtable has achieved several goals with applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Wallet, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented bulk processing jobs to latency-sensitive serving of data to end users.

Bigtable combines usability and flexibility with scalable performance. Its flexibility and usability are due to a simple 10 principles model that governs how data is structured and accessed.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp, each of which is in the same unstructured array of bytes:

```
   (key_prefix, column_family, timestamp) → string
```

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JI Furman, Andrey Khoriin, James Larson, Jean-Michel Leon, Yves Le, Alexander Lloyd, Vitaly Vasserman

Google Inc.

ABSTRACT

Megastore is a storage system developed to meet the requirements of today’s interactive online services. Megastore blends the scalability of a RDBMS database with the economics of a traditional BDBSM in a novel way, and provides both strong consistency guarantees and high scalability. We provide a low-level scalable ACID interface within and across machines in a cluster, which allows applications to seamlessly exploit each node to access a wide-area network with reasonable latency and support massive failures between databases. This paper describes Megastore’s in-memory and replicated object storage. It also describes our experiences supporting a wide range of Google production services built with Megastore.

Categories and Subject Descriptors

C.2.1 [Database Management]: Systems—concurrent, distributed systems

General Terms

Algorithms, Design, Performance, Reliability

Keywords

Large-dimension, Distributed transaction, Bigtable, Paxos

1. INTRODUCTION

Interactive web services are facing the storage challenges to meet new demands as interactive applications are becoming more rich and complex, and as more and more data is generated by the Internet. The most common way to store data for these services is a distributed key-value store with the semantics of a traditional BDBSM. A key concern is how to ensure the availability of service at large scale, in an environment where the failure of individual data nodes, machines, or routers all the way up to large-scale outages affecting entire clusters.

These requirements are in conflict. Relational databases provide a rich set of features and strong consistency guarantees. Key-value stores provide high availability, but lose other properties like transactional consistency and data durability. The key-value store is challenged as it is performing a consistent view of replicated data, especially during failures.

Megastore is a storage system developed to meet the storage requirements of today’s interactive online services. It provides a scalable ACID interface to applications that store and retrieve data at large scale while providing strong consistency guarantees and high availability.

This article is published under a Creative Commons Attribution License (CC BY) which permits derivative works and redistributions provided the original work is properly cited. Please see the license text online for a full description of terms.

6th Biennial Conference on Innovative Data Systems Research (CIDR ’11)
Memoirs of Innovative Data Systems Research (CIDR ’11) Symposium V. 6, 2011, Victoria, Canada, USA
GOOGLE SPANNER

Google’s geo-replicated DBMS (>2011)
Schematized, semi-relational data model.
Decentralized shared-disk architecture.
Log-structured on-disk storage.
Concurrency Control:
→ Strict 2PL + MVCC + Multi-Paxos + 2PC
→ Externally consistent global write-transactions with synchronous replication.
→ Lock-free read-only transactions.
CREATE TABLE users {
  uid INT NOT NULL,
  email VARCHAR,
  PRIMARY KEY (uid)
};
CREATE TABLE albums {
  uid INT NOT NULL,
  aid INT NOT NULL,
  name VARCHAR,
  PRIMARY KEY (uid, aid)
} INTERLEAVE IN PARENT users
ON DELETE CASCADE;
CREATE TABLE users {
  uid INT NOT NULL,
  email VARCHAR,
  PRIMARY KEY (uid)
};

CREATE TABLE albums {
  uid INT NOT NULL,
  aid INT NOT NULL,
  name VARCHAR,
  PRIMARY KEY (uid, aid)
} INTERLEAVE IN PARENT users ON DELETE CASCADE;

Physical Storage

| users(1001) |
| albums(1001, 9990) |
| albums(1001, 9991) |
| users(1002) |
| albums(1002, 6631) |
| albums(1002, 6634) |
CONCURRENCY CONTROL

MVCC + Strict 2PL with Wound-Wait Deadlock Prevention

Ensures ordering through globally unique timestamps generated from atomic clocks and GPS devices.

Database is broken up into tablets:
→ Use Paxos to elect leader in tablet group.
→ Use 2PC for txns that span tablets.
SPANNER TABLETS

Tablet A

Data Center 1

Paxos

Tablet A

Data Center 2

Leader

Tablet A

Data Center 3

Paxos

Writes + Reads
SPANNER TABLETS

Snapshot Reads

Paxos

Tablet A

Data Center 1

Writes + Reads

Paxos

Tablet A

Data Center 2

Snapshot Reads

Paxos

Tablet A

Data Center 3

Leader
SPANNER TABLETS

Snapshots Reads

Tablet A
Data Center 1

Paxos Groups

Tablet A
Data Center 2

Leader

Tablet A
Data Center 3

Paxos

2PC

Tablet B-Z

Paxos

Write + Reads

Snapshot Reads

Paxos Groups
Spanner orders transactions based on physical "wall-clock" time.
→ This is necessary to guarantee linearizability.
→ If $T_1$ finishes before $T_2$, then $T_2$ should see the result of $T_1$.

Each Paxos group decides in what order transactions should be committed according to the timestamps.
→ If $T_1$ commits at $\text{time}_1$ and $T_2$ starts at $\text{time}_2 > \text{time}_1$, then $T_1$'s timestamp should be less than $T_2$'s.
The DBMS maintains a global wall-clock time across all data centers with bounded uncertainty. Timestamps are intervals, not single values.
SPANNER TRUETIME

Acquire Locks
Commit Timestamp
$s > \text{TT.now().latest}$

Commit +
Release Locks
Wait until
$s > \text{TT.now().earliest}$

TIME

Commit Wait
average $\epsilon$
average $\epsilon$
OCC engine built on top of Spanner.

→ In the read phase, F1 returns the last modified timestamp with each row. No locks.

→ The timestamp for a row is stored in a hidden lock column. The client library returns these timestamps to the F1 server.

→ If the timestamps differ from the current timestamps at the time of commit the transaction is aborted.
Spanner Database-as-a-Service. AFAIK, it is based on Spanner SQL not F1.
Distributed **document** DBMS started in 2007.
→ Document → Tuple
→ Collection → Table/Relation

Open-source (Server Side Public License)

Centralized shared-nothing architecture.

Concurrency Control:
→ OCC with multi-granular locking
A customer has orders and each order has order items.

- **Customers** → $R_1(\text{custId, name, ...})$
- **Orders** → $R_2(\text{orderId, custId, ...})$
- **Order Items** → $R_3(\text{itemId, orderId, ...})$
A customer has orders and each order has order items.
A customer has orders and each order has order items.

```json
{
  "custId": 1234,
  "custName": "Andy",
  "orders": [
    {
      "orderId": 9999,
      "orderItems": [
        {
          "itemId": "XXXX",
          "price": 19.99
        },
        {
          "itemId": "YYYY",
          "price": 29.99
        }
      ]
    }
  ]
}
```
QUERY EXECUTION

JSON-only query API

No cost-based query planner / optimizer. → Heuristic-based + "random walk" optimization.

JavaScript UDFs (not encouraged).

Supports server-side joins (only left-outer?).

Multi-document transactions (new in 2018).
DISTRIBUTED ARCHITECTURE

Heterogeneous distributed components.
→ Shared nothing architecture
→ Centralized query router.

Master-slave replication.

Auto-sharding:
→ Define 'partitioning' attributes for each collection (hash or range).
→ When a shard gets too big, the DBMS automatically splits the shard and rebalances.
**MongoDB Cluster Architecture**

- **Application Server**
- **Config Server (mongod)**
- **Router (mongos)**
- **Shards (mongod)**

Diagram:
- Application Server
- Config Server
- Router (mongos)
- Shards (mongod)

Diagram details:
- Get Id=101
- P1, P2, P3, P4
MONGODB CLUSTER ARCHITECTURE

Router (mongos)

Shards (mongod)

Router (mongos)

Get Id=101

Config Server (mongod)

Application Server

P1→ID: 1-100
P2→ID: 101-200
P3→ID: 201-300
P4→ID: 301-400
MONGODB CLUSTER ARCHITECTURE

Router (mongos)

Shards (mongod)

Application Server

Get Id=101

Config Server (mongod)

Router (mongos)

P1→ID: 1-100
P2→ID: 101-200
P3→ID: 201-300
P4→ID: 301-400
STORAGE ARCHITECTURE

Originally used mmap storage manager
→ No buffer pool.
→ Let the OS decide when to flush pages.
→ Single lock per database.

MongoDB v3 supports pluggable storage backends
→ WiredTiger from BerkeleyDB alumni.
   http://cmudb.io/lectures2015-wiredtiger
→ RocksDB from Facebook (“MongoRocks”)
   http://cmudb.io/lectures2015-rocksdb
ANDY'S CONCLUDING REMARKS

Databases are awesome.
→ They cover all facets of computer science.
→ We have barely scratched the surface...

Going forth, you should now have a good understanding how these systems work.

This will allow you to make informed decisions throughout your entire career.
→ Avoid premature optimizations.