Final Review + Systems Potpourri
ADMINISTRIVIA

Project #4: TODAY @ 11:59pm
FINAL EXAM

**Who:** You

**What:** [http://cmudb.io/f17-final](http://cmudb.io/f17-final)

**When:** Friday Dec 15th @ 5:30pm

**Where:** GHC 4401

**Why:** Because otherwise Joy and I call your family over the break...
FINAL EXAM

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

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

Andy:
→ Wednesday Dec. 13th @ 12:00pm-1:30pm
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
TRANSACTIONS

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

B+Tree Latch Crabbing

Locks vs. Latches
Transactions

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></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>32</td>
<td>Google Spanner/F1</td>
</tr>
<tr>
<td>Google Spanner/F1</td>
<td>22</td>
<td>MongoDB</td>
</tr>
<tr>
<td>LinkedIn Espresso</td>
<td>16</td>
<td>CockroachDB</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>16</td>
<td>Apache Hbase</td>
</tr>
<tr>
<td>Facebook Scuba</td>
<td>16</td>
<td>Peloton</td>
</tr>
<tr>
<td>Apache Hbase</td>
<td>14</td>
<td>Facebook Scuba</td>
</tr>
<tr>
<td>VoltDB</td>
<td>10</td>
<td>Cloudera Impala</td>
</tr>
<tr>
<td>Redis</td>
<td>10</td>
<td>Apache Hive</td>
</tr>
<tr>
<td>Vertica</td>
<td>5</td>
<td>Apache Cassandra</td>
</tr>
<tr>
<td>Cloudera Impala</td>
<td>5</td>
<td>LinkedIn Espresso</td>
</tr>
</tbody>
</table>
Bigtable: A Distributed Storage System for Structured Data
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, William C. Hsieh, Deborah A. Wallach
Mike Burrows, Tesla Chandra, Andrew Fikes, Robert E. Gruber

Google, Inc.

Abstract
Bigtable is a distributed storage system managed by Google. Bigtable manages data in a non-blocking fashion, with no wait or call to databases. Many projects at Google store data in Bigtable, including web-indexing, Google Earth, and Google Maps. Applications place different demands on Bigtable. While it is a centralized data index (either URL or Google Earth), the project is designed to scale out to very large storage devices. We present a simple model of a distributed system that uses dynamic control over data layout and format, and allows clients to reason about the locality properties of the data themselves in the underlying storage. Data is indexed using row and column names that are available in queries.

1 Introduction
Over the last two and a half years we have designed and implemented a distributed storage system for managing structured data called Google Bigtable. Bigtable is designed to be scalable to petabytes of data and thousands of machines. It has achieved several goals: high availability, scalability, liveliness, and high availability. Bigtable is used by a number of other projects, including Bigtable Analytics, Google Finance, Okapi, Personalized Search, Wikipedia, and Google Earth. These projects use Bigtable for a variety of demanding workloads, which range from long-standing distributed processing tasks to large scale service-oriented applications. The Bigtable overview of these tasks spans a wide range of applications, from a handful to thousands of servers, and more than a few thousand billion-points of data. In this paper, we will present the overview of a distributed system that provides an integrated environment for managing structured data.

2 Data Model
A Bigtable is a sparse, distributed, persistent multi-dimensional database. The map is indexed by a row key, column key, and a timestamp; each value in the map is stored in a multi-dimensional array of bytes.

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Google, Inc.

ABSTRACT
Megastore is a storage system designed and implemented to provide scalable and reliable support for interactive applications where the data is not necessarily small and the write workload is relatively high. Megastore is designed to provide a highly available, scalable, persistent, and reliable distributed system that supports interactive application developers using highly available, scalable, and consistent support for large amounts of data. Megastore employs high-capacity RAID arrays to store the data, and a highly available, scalable, and consistent system to provide reliable service with low latency.

Categories and Subject Descriptors
C.2.3 [Database Management]: Distributed databases, 6.4.4 [Distributed Management]: System configuration, distributed storage systems

General Terms
Algorithms, Design, Performance, Reliability

Keywords
Large databases, Distributed transactions, Bigtable, Paean, MonetDB, Cassandra

1. Introduction
Interactive online services are facing the storage constraints to meet user demands as the volume of data grows. Megastore is a storage system designed to meet the needs of interactive applications where the need for interactive responses to user input is paramount. Megastore is designed to provide a highly available, scalable, and consistent system to support interactive applications.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. If a work is covered by the Creative Commons license, the terms of the license must be obeyed.

Copyright © 2012 ACM, Inc.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This article is published by the Computer Science Department of the University of California, Berkeley, which licenses publication rights, and is subject to the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
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
→ Externally consistent global write-transactions with synchronous replication.
→ Lock-free read-only transactions.
## PHYSICAL DENORMALIZATION

```sql
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

<table>
<thead>
<tr>
<th>Table</th>
<th>Primary Key</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>users(1001)</td>
<td></td>
<td>albums(1001, 9990)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>albums(1001, 9991)</td>
</tr>
<tr>
<td>users(1002)</td>
<td></td>
<td>albums(1002, 6631)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>albums(1002, 6634)</td>
</tr>
</tbody>
</table>
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;

<table>
<thead>
<tr>
<th>Physical Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>users(1001)</strong></td>
</tr>
<tr>
<td>&gt;albums(1001, 9990)</td>
</tr>
<tr>
<td>&gt;albums(1001, 9991)</td>
</tr>
<tr>
<td><strong>users(1002)</strong></td>
</tr>
<tr>
<td>&gt;albums(1002, 6631)</td>
</tr>
<tr>
<td>&gt;albums(1002, 6634)</td>
</tr>
</tbody>
</table>
PAXOS

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.

→ First correct protocol that was provably resilient in the face asynchronous networks
Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed. Does not block if a majority of participants are available and has provably minimal message delays in the best case.

→ First correct protocol that was provably resilient in the face asynchronous networks
PAXOS

Application Server

Proposer

Commit Request

Node 1

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Commit Request

Propose

Node 2

Node 3

Node 4

Acceptors
PAXOS

Application Server

Proposer

Node 1

Commit Request

Propose

Node 2

X

Node 3

Acceptors

Node 4
PAXOS

Commit Request

Node 1

Propposer

Commit

Node 2

Agree

Node 3

Node 4

Agree

Acceptors

Application Server
Success!
PAXOS

Propose(n) → Agree(n) → Commit(n) → Propose(n+1)
PAXOS

Proposer

Propose(n)

Agree(n)

Commit(n)

Reject(n,n+1)

Proposer

Propose(n+1)

TIME
PAXOS

- Propose(n)
- Agree(n)
- Propose(n+1)
- Commit(n)
- Reject(n,n+1)
- Agree(n+1)
PAXOS

Propose(n) -> Agree(n) -> Propose(n+1)

Commit(n) -> Reject(n,n+1) -> Agree(n+1) -> Commit(n+1) -> Accept(n+1)

TIME
MULTI-PAXOS

If the system elects a single leader that is in charge of proposing changes for some period of time, then it can skip the PREPARE phase.
→ Fall back to full Paxos whenever there is a failure.

The system has to periodically renew who the leader is.
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

Data Center 1

Data Center 2

Leader

Data Center 3

Paxos Group
SPANNER TABLETS

Data Center 1

Data Center 2

Leader

Data Center 3

Paxos Group

Writes + Reads
SPANNER TABLETS

Paxos Group

Data Center 1

Data Center 2

Leader

Data Center 3

writes + reads

Paxos

Paxos
SPANNER TABLETS

Snapshot Reads → Tablet → Data Center 1

Writes + Reads → Tablet → Data Center 2 (Leader)

Snapshot Reads → Tablet → Data Center 3
TRANSACTION ORDERING

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 $\text{time}_2$ where $\text{time}_1 < \text{time}_2$, 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.

```
TT.now()
```

earliest  latest
The DBMS maintains a global wall-clock time across all data centers with bounded uncertainty.

Timestamps are intervals, not single values
SPANNER TRUETIME

Each data center has GPS and atomic clocks
→ These two provide fine-grained clock synchronization down to a few milliseconds.
→ Every 30 seconds, there's maximum 7 ms difference.

Multiple sync daemons per data center
→ GPS and atomic clocks can fail in various conditions.
→ Sync daemons talk to each other within a data center as well as across data centers.
SPANNER TRUETIME

Acquire Locks

TIME
SpANNER TRUETIME

Acquire Locks

Commit Timestamp

s > TT.now().latest

s
SPANNER TRUE TIME

Acquire Locks

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

$\text{TT.now().earliest} > s$

Wait until
SPANNER TRUE TIME

Acquire Locks

Commit Timestamp
s > TT.now().latest

TIME

s

Wait until
TT.now().earliest > s

Commit Wait

average ε

average ε
SPANNER TRUE TIME

- Acquire Locks
- Commit Timestamp \( s > \text{T.T.now().latest} \)
- \( s \) (Commit Wait)
- Commit + Release Locks
- Wait until \( \text{T.T.now().earliest} > s \)

\[ \text{average } \varepsilon \quad \text{average } \varepsilon \]
GOOGLE F1 (2013)

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.
MONGODB

Document Data Model
→ Think JSON, XML, Python dicts
→ Not Microsoft Word documents

Different terminology:
→ Document ➔ Tuple
→ Collection ➔ Table/Relation
BCNF EXAMPLE

A customer has orders and each order has order items.
BCNF EXAMPLE

A customer has orders and each order has order items.

Customers → R₁(custId, name, ...)

Orders → R₂(orderId, custId, ...)

Order Items → R₃(itemId, orderId, ...)

CMU 15-445/645 (Fall 2017)
A customer has orders and each order has order items.
BCNF EXAMPLE

A customer has orders and each order has order items.
BCNF EXAMPLE

A customer has orders and each order has order items.
BCNF EXAMPLE

A customer has orders and each order has order items.

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

JSON-only query API
Single-document atomicity.
→ **OLD**: No server-side joins. Had to "pre-join" collections by embedding related documents inside of each other.
→ **NEW**: Server-side joins (only left-outer equi)

No cost-based query planner / optimizer.
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

Router (mongos)

Router (mongos)

Config Server (mongod)

Shards (mongod)

P1

P2

P3

P4
MONGODB CLUSTER ARCHITECTURE

Router (mongos)

Router (mongos)

Config Server (mongod)

Shards (mongod)

Application Server

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

Application Server

Router (mongos)

Router (mongos)

Config Server (mongod)

Shards (mongod)

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.

→ **WiredTiger** from BerkeleyDB alumni.
→ **RocksDB** from Facebook (“MongoRocks”)
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 career.
→ Both MySQL and Postgres are getting really good...
→ Avoid premature optimizations.