Who: You
What: Final Exam
Where: Gradescope + OHQueue + Google Doc
When: Thu Dec 17\textsuperscript{th} (Two Sessions)
Why: https://youtu.be/yCotpBAqJho

https://15445.courses.cs.cmu.edu/fall2020/final-guide.html
FINAL EXAM

Two Exam Sessions:
→ Session #1: Thu Dec 17th @ 8:30am ET
→ Session #2: Thu Dec 17th @ 8:00pm ET
→ I will email you to confirm your session.

Exam will be available on Gradescope.

Please email Andy if you need special accommodations.
**FINAL EXAM**

Exam covers all lecture material in the entire course but will emphasize topics after mid-term.

Open book/notes/calculator.

You are **not** required to turn on your video during the video.

We will answer clarification questions via OHQ and post announcements on a Google Doc.
COURSE EVALS

Your feedback is strongly needed:
→ https://cmu.smartevals.com
→ https://www.ugrad.cs.cmu.edu/ta/F20/feedback/

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

Andy's hours:
→ Monday Dec 14th @ 3:20-4:40pm
→ Mon Dec 14th + Wed Dec 16th @ 10pm: [link](https://calendly.com/andy-pavlo/f20-andy-after-dark)
→ Or by appointment

All TAs will have their regular office hours up to and including Saturday Dec 12th
STUFF BEFORE MID-TERM

SQL
Buffer Pool Management
Hash Tables
B+Trees
Storage Models
Inter-Query Parallelism
ACID
Conflict Serializability:
→ How to check?
→ How to ensure?
View Serializability
Recoverable Schedules
Isolation Levels / Anomalies
TRANSACTIONS

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

Multiple Granularity Locking
→ Intention Locks
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>2018</th>
<th></th>
<th>2019</th>
<th></th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cockroach LABS</td>
<td>26</td>
<td>Scuba</td>
<td>20</td>
<td>Amazon DynamoDB</td>
<td>18</td>
</tr>
<tr>
<td>Google Spanner</td>
<td>25</td>
<td>mongoDB</td>
<td>19</td>
<td>cassandra</td>
<td>15</td>
</tr>
<tr>
<td>mongoDB</td>
<td>24</td>
<td>Cockroach LABS</td>
<td>18</td>
<td>mongoDB</td>
<td>14</td>
</tr>
<tr>
<td>Amazon Aurora</td>
<td>18</td>
<td>Amazon Aurora</td>
<td>17</td>
<td>elasticsearch</td>
<td>13</td>
</tr>
<tr>
<td>redis</td>
<td>18</td>
<td>Google Spanner</td>
<td>17</td>
<td>PostgreSQL</td>
<td>13</td>
</tr>
<tr>
<td>cassandra</td>
<td>17</td>
<td>snowflake</td>
<td>17</td>
<td>Amazon Aurora</td>
<td>12</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>12</td>
<td>PostgreSQL</td>
<td>17</td>
<td>SQLite</td>
<td>11</td>
</tr>
<tr>
<td>Hive</td>
<td>11</td>
<td>OceanBase</td>
<td>16</td>
<td>MySQL</td>
<td>11</td>
</tr>
<tr>
<td>Scuba</td>
<td>10</td>
<td>Amazon REDSHIFT</td>
<td>15</td>
<td>splunk</td>
<td>9</td>
</tr>
<tr>
<td>MySQL</td>
<td>10</td>
<td>elasticsearch</td>
<td>15</td>
<td>RocksDB</td>
<td>9</td>
</tr>
</tbody>
</table>
Amazon publishes a paper in 2007 on the original Dynamo system.
→ Eventually consistency key/value store
→ Shared-nothing architecture
→ Non-SQL API, no joins, no transactions
→ Partitions based on consistent hashing

Amazon makes DynamoDB available to customers on AWS in 2012.

HISTORY

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandio, Dinesh Mantravadi, Madison Apley, Abhishek Upadhyay, and Werner Vogels

ABSTRACT

At Amazon, as in most e-commerce sites, the large scale and the variety of services being offered to its customers makes managing a robust, scalable, and highly available database a challenging task. A key challenge is providing access to a wide variety of data with a high degree of flexibility. In the past, the database infrastructure has been based on the relational model, which has provided satisfactory performance but has limited flexibility and scalability. The Amazon database team was charged with developing a system that could support the needs of Amazon’s e-commerce business. The system was to be highly available and maintain consistent reads and eventually consistent writes. The system would allow any service to store and access data with a simple API, without having to deal with complex issues surrounding data consistency. The Dynamo database system, described in this paper, provides a simple, high-level interface to a highly available key-value store.

Keywords: key-value store, consistency, flexible, scalable, reliability

1. INTRODUCTION

At Amazon, as in most e-commerce sites, the large scale and the variety of services being offered to its customers makes managing a robust, scalable, and highly available database a challenging task. A key challenge is providing access to a wide variety of data with a high degree of flexibility. In the past, the database infrastructure has been based on the relational model, which has provided satisfactory performance but has limited flexibility and scalability. The Amazon database team was charged with developing a system that could support the needs of Amazon’s e-commerce business. The system was to be highly available and maintain consistent reads and eventually consistent writes. The system would allow any service to store and access data with a simple API, without having to deal with complex issues surrounding data consistency. The Dynamo database system, described in this paper, provides a simple, high-level interface to a highly available key-value store.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandio, Dinesh Mantravadi, Madison Apley, Abhishek Upadhyay, and Werner Vogels

ABSTRACT

At Amazon, as in most e-commerce sites, the large scale and the variety of services being offered to its customers makes managing a robust, scalable, and highly available database a challenging task. A key challenge is providing access to a wide variety of data with a high degree of flexibility. In the past, the database infrastructure has been based on the relational model, which has provided satisfactory performance but has limited flexibility and scalability. The Amazon database team was charged with developing a system that could support the needs of Amazon’s e-commerce business. The system was to be highly available and maintain consistent reads and eventually consistent writes. The system would allow any service to store and access data with a simple API, without having to deal with complex issues surrounding data consistency. The Dynamo database system, described in this paper, provides a simple, high-level interface to a highly available key-value store.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Dynamo: Amazon’s Highly Available Key-value Store

Giuseppe DeCandio, Dinesh Mantravadi, Madison Apley, Abhishek Upadhyay, and Werner Vogels

ABSTRACT

At Amazon, as in most e-commerce sites, the large scale and the variety of services being offered to its customers makes managing a robust, scalable, and highly available database a challenging task. A key challenge is providing access to a wide variety of data with a high degree of flexibility. In the past, the database infrastructure has been based on the relational model, which has provided satisfactory performance but has limited flexibility and scalability. The Amazon database team was charged with developing a system that could support the needs of Amazon’s e-commerce business. The system was to be highly available and maintain consistent reads and eventually consistent writes. The system would allow any service to store and access data with a simple API, without having to deal with complex issues surrounding data consistency. The Dynamo database system, described in this paper, provides a simple, high-level interface to a highly available key-value store.
CONSISTENT HASHING

hash(key1)

hash(key2)
CONSISTENT HASHING

If hash(key) = P4
CONSISTENT HASHING
CONSISTENT HASHING

Replication Factor = 3

hash(key1)
DynamoDB supports a subset of the relational model:
→ A table definition includes the set of attributes that the table's records must contain.
→ You cannot specify constraints (integrity, foreign key).

Tables support two types of primary keys:
→ Single Partition Key
→ **Composite Partition Key + Sort Key**
DYNAMODB SECONDARY INDEXES

Local Secondary Index
→ Use the partition key to initially route the request to a node.
→ Each node maintains a local B+Tree that only contains keys for the records stored at that node.

Global Secondary Index
→ Uses a partition key + sort key that is different than the table's primary key.
→ Not guaranteed to be consistent with the table.
Access the database using API calls.
→ Application must perform additional operations client-side if they need functionality beyond provided API.

Modification API calls support application-specified **conditionals** to deal with eventual consistency issues.
DYNAMODB TRANSACTIONS

In 2018, Amazon announced support for client-side transaction support in DynamoDB.

→ Centralized Middleware Coordinator

Called "single-shot" transactions because you need to know your read/write set before the transaction starts.

TransactionGetItems(
  Get(table:T1, key:k1),
  Get(table:T2, key:k2),
  Get(table:T3, key:k3)
)

TransactionWriteItems(
  Put(table:T1, key:k1, value:v1),
  Delete(table:T2, key:k2),
  Update(table:T3, key:k3),
  Check(table:T3, key:k3, value:<100)
)

Source: Doug Terry
DYNAMODB TRANSACTIONS

Move $100 from Andy's bank account to his promotor's account.

BEGIN
A = A - 100
B = B + 100
COMMIT

A = Get(person: "Andy")
B = Get(person: "Bookie")
TransactionWriteItems(
    Check(person: "Andy", balance: A),
    Check(person: "Bookie", balance: B),
    Put(person: "Andy", balance: A-100),
    Put(person: "Bookie", balance: B+100)
)
After Amazon published the Dynamo paper in 2007, people at Facebook start writing a clone called Cassandra in 2008 for their message service. → Decided to not use the DBMS and instead released the source code.

Picked up by organizations outside of Facebook and then became an Apache project in 2009.
APACHE CASSANDRA

Borrows a lot of ideas from other systems:
→ Eventual Consistency
→ Shared-Nothing
→ Consistent Hashing (Amazon Dynamo)
→ Column-Family Data Model (Google BigTable)
→ Log-structured Merge Trees

Originally one of the main proponents of the NoSQL movement but now pushing CQL.
The log is the database.
→ DBMS reads log to reconstruct the record for a read.

**MemTable**: In-memory cache

**SSTables**:
→ Read-only portions of the log.
→ Use indexes + Bloom filters to speed up reads
→ See the CMU-DB **RocksDB** talk (2015)
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 one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.
A **CUSTOMER** has one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.
A **CUSTOMER** has one or more **ORDER** records. Each **ORDER** record has one or more **ORDER_ITEM** records.
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.
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

Router (mongos)

Shards (mongod)

P1

P2

P3

P4

Config Server (mongod)

Application Server

Get Id=101

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.
  

→ **RocksDB** from Facebook (“MongoRocks”)
  
MANGODB

Single-node satirical implementation of MongoDB written in Python.
→ Only supports MongoDB wire protocol v2

All data is written to /dev/null

https://github.com/dcramer/mangodb
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.