Logistics

• HW6 is due tomorrow (Thursday, Apr 28).

• MP3 is due next week, May 5th.

• Final exam on May 11th
  • Please register on CBTF.
  • Same format as midterms, but longer (3 hours).
  • **Comprehensive:** includes everything covered in the course.
  • ~50% weightage assigned to materials that were not covered by midterm 1 and midterm 2 syllabus (i.e. blockchains and beyond).
# Grade distribution

<table>
<thead>
<tr>
<th></th>
<th>3-credit</th>
<th>4-credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homework</td>
<td>33%</td>
<td>16% (drop 2 worst HWs)</td>
</tr>
<tr>
<td>Midterms</td>
<td>33%</td>
<td>25%</td>
</tr>
<tr>
<td>Final</td>
<td>33%</td>
<td>25%</td>
</tr>
<tr>
<td>MPs</td>
<td>N/A</td>
<td>33%</td>
</tr>
<tr>
<td>Participation</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Grading

• Midterm curving formula (tentative)
  • absolute: \( 100 \times \frac{\text{your score}}{\text{total score}} \)
  • relative: \( 80 + 10\times\frac{\text{your score} - \text{avg}_\text{UG}\_\text{score}}{\text{standard}_{\text{dev}}^\text{dev}} \)
  • We will use max(absolute, relative) to get final score out of 100.

• Midterm 1:
  • avg\_UG\_score = 55.43 (out of 70)
  • standard\_dev = 8.24

• Midterm 2:
  • avg\_UG\_score = 43.13 (out of 65)
  • standard\_dev = 9.72

• Multiply the final score (out of 100) for each midterm by:
  • 0.165 for 3-credit students
  • 0.125 for 4-credit students.

• Finals will be similarly curved, but has higher weightage.
Grading

• Homeworks will not be curved.
  • For 3-credit students:
    • (sum of all 6 homework scores) * 100 * 0.33 / 240
  • For 4-credit students:
    • (sum of best 4 homework scores) * 100 * 0.16 / 160

• MPs will not be curved.
  • (sum of all four MP scores) * 100 * 0.33 / 330

• Participation score: directly taken from Campuswire
  • if reported score > 100, you get full 1%
  • Else you get (reported score /100)%
Final Grades

- **Tentative mapping from score to grade** (*rough estimate*):
  - Cutoff for B: 80%
  - Bump up a grade for each 4% leap above 80%.
    - B+ 84%, A- 88%, A 92%, A+ 96%
  - Bump down a grade for each 4% leap below 80%
    - B- 76%, C+ 72%, …..

- This is subject to change!
Our agenda

• Brief overview of key-value stores

• Distributed Hash Tables
  • Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

• Key-value stores in the cloud
  • How to run large-scale distributed computations over key-value stores?
    • Map-Reduce Programming Abstraction
  • How to design a large-scale distributed key-value store?
    • Case-study: Facebook’s Cassandra
Today’s focus

• Brief overview of key-value stores

• Distributed Hash Tables
  • Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.

• Key-value stores in the cloud
  • How to run large-scale distributed computations over key-value stores?
    • Map-Reduce Programming Abstraction
  • How to design a large-scale distributed key-value store?
    • Case-study: Facebook’s Cassandra
Distributed datastores

• Distributed datastores
  • Service for managing distributed storage.

• Distributed NoSQL key-value stores
  • BigTable by Google
  • HBase open-sourced by Yahoo and used by Hadoop.
  • DynamoDB by Amazon
  • Cassandra by Facebook
  • Voldemort by LinkedIn
  • MongoDB,
  • …

• Spanner is not a NoSQL datastore. It’s more like a distributed relational database.
Key-value/NoSQL Data Model

• NoSQL = “Not Only SQL”
• Necessary API operations: `get(key)` and `put(key, value)`
  • And some extended operations, e.g., “CQL” in Cassandra key-value store

• Tables
  • Like RDBMS tables, but …
  • May be unstructured: May not have schemas
    • Some columns may be missing from some rows
  • Don’t always support joins or have foreign keys
  • Can have index tables, just like RDBMSs
How to design a distributed key-value datastore?
Design Requirements

- High performance, low cost, and scalability.
  - Speed (high throughput and low latency for read/write)
  - Low TCO (total cost of operation)
- Fewer system administrators
- Incremental scalability
  - Scale out: add more machines.
  - Scale up: upgrade to powerful machines.
  - *Cheaper to scale out than to scale up.*
Design Requirements

• High performance, low cost, and scalability.

• Avoid single-point of failure
  • Replication across multiple nodes.

• Consistency: reads return latest written value by any client (all nodes see same data at any time).
  • Different from the C of ACID properties for transaction semantics!

• Availability: every request received by a non-failing node in the system must result in a response (quickly).
  • Follows from requirement for high performance.

• Partition-tolerance: the system continues to work in spite of network partitions.
Theorem

- **Consistency**: reads return latest written value by any client (all nodes see same data at any time).
- **Availability**: every request received by a non-failing node in the system must result in a response (quickly).
- **Partition-tolerance**: the system continues to work in spite of network partitions.

- In a distributed system you can only guarantee at most 2 out of the above 3 properties.
  - Proposed by Eric Brewer (UC Berkeley)
  - Subsequently proved by Gilbert and Lynch (NUS and MIT)
CAP Theorem

- Data replicated across both N1 and N2.
- If network is partitioned, N1 can no longer talk to N2.
- Consistency + availability require N1 and N2 must talk.
  - no partition-tolerance.
- Partition-tolerance + consistency:
  - only respond to requests received at N1 (no availability).
- Partition-tolerance + availability:
  - write at N1 will not be captured by a read at N2 (no consistency).
CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability.

Conventional RDBMSs (non-replicated)

Consistency

HBase, HyperTable, BigTable, Spanner

Partition-tolerance

Cassandra, RIAK, Dynamo, Voldemort

Availability
Modern key-value stores vs. RDBMS

• While RDBMS provide ACID
  • Atomicity
  • Consistency
  • Isolation
  • Durability

• Many modern key-value stores provide BASE
  • Basically Available Soft-state Eventual Consistency
  • Prefers Availability over Consistency
Case Study: Cassandra
Cassandra

• A distributed key-value store.
• Intended to run in a datacenter (and also across DCs).
• Originally designed at Facebook.
• Open-sourced later, today an Apache project.
• Some of the companies that use Cassandra in their production clusters.
  • IBM, Adobe, HP, eBay, Ericsson, Symantec
  • Twitter, Spotify
  • PBS Kids
  • Netflix: uses Cassandra to keep track of your current position in the video you’re watching
Data Partitioning: Key to Server Mapping

• How do you decide which server(s) a key-value resides on?

Cassandra uses a ring-based DHT but without finger or routing tables.

Say $m=7$

Client

Coordinator

Read/write K13

Primary replica for key K13

Backup replicas for key K13

One ring per DC
Partitioner

- Component responsible for key to server mapping (hash function).

- Two types:
  - Chord-like hash partitioning
    - Murmer3Partitioner (default): uses murmer3 hash function.
    - RandomPartitioner: uses MD5 hash function.
  - ByteOrderedPartitioner: Assigns ranges of keys to servers.
    - Easier for range queries (e.g., get me all twitter users starting with [a-b])

- Determines the primary replica for a key.
Replication Policies

Two options for replication strategy:

1. **SimpleStrategy:**
   - First replica placed based on the partitioner.
   - Remaining replicas clockwise in relation to the primary replica.

2. **NetworkTopologyStrategy:** for multi-DC deployments
   - Two or three replicas per DC.
   - Per DC
     - First replica placed according to Partitioner.
     - Then go clockwise around ring until you hit a different rack.
**Writes**

- Need to be lock-free and fast (no reads or disk seeks).

- Client sends write to one coordinator node in Cassandra cluster.
  - Coordinator may be per-key, or per-client, or per-query.

- Coordinator uses Partitioner to send query to all replica nodes responsible for key.

- When X replicas respond, coordinator returns an acknowledgement to the client
  - $X = \text{any one, majority, all} \ldots \text{(consistency spectrum)}$
  - More details later!
Writes: Hinted Handoff

• Always writable: **Hinted Handoff mechanism**
  • If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
  • When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).
Writes at a replica node

On receiving a write

1. Log it in disk commit log (for failure recovery)

2. Make changes to appropriate memtables
   - **Memtable** = In-memory representation of multiple key-value pairs
   - Cache that can be searched by key
   - Write-back cache as opposed to write-through

3. Later, when memtable is full or old, flush to disk
   - Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
   - Index file: An SSTable of (key, position in data sstable) pairs
   - And a Bloom filter (for efficient search) – next slide.
To be continued in next class

- Wrap up writes.
- Reads.
- Cluster membership.
- Eventual consistency model.