NoSQL and Key-Value Stores

CS425/ECE428—SPRING 2019
NIKITA BORISOV, UIUC
Relational Databases

Row-based table structure
- Well-defined *schema*
- Complex queries using JOINS
  SELECT Firstname, Lastname
  FROM Students
  JOIN Enrollment on Students.UIN == Enrollment.UIN
  WHERE Enrollment.CRN = 37205

Transactional semantics
- Atomicity
- Consistency
- Integrity
- Durability

<table>
<thead>
<tr>
<th>UIN</th>
<th>First name</th>
<th>Last name</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>John</td>
<td>Smith</td>
<td>CS</td>
</tr>
<tr>
<td>1256</td>
<td>Alice</td>
<td>Jones</td>
<td>ECE</td>
</tr>
<tr>
<td>1357</td>
<td>Jane</td>
<td>Doe</td>
<td>PHYS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CRN</th>
<th>Dept</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>37205</td>
<td>ECE</td>
<td>428</td>
</tr>
<tr>
<td>37582</td>
<td>CS</td>
<td>425</td>
</tr>
<tr>
<td>35724</td>
<td>PHYS</td>
<td>212</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CRN</th>
<th>UIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>37205</td>
<td>1234</td>
</tr>
<tr>
<td>37582</td>
<td>1256</td>
</tr>
<tr>
<td>35724</td>
<td>1357</td>
</tr>
</tbody>
</table>
Distributed Transactions

Participants ensure isolation using two-phase locking

Coordinator ensures atomicity using two—phase commit

Replica managers ensure availability / durability
  ◦ Quorums ensure one-copy serializability

Locking can be expensive
  ◦ SELECT query can grab read lock on entire table

2PC latency is high
  ◦ Two round-trips in addition to base transaction overhead
  ◦ Runs at the speed of slowest participant
    ◦ (Which runs at the speed of the slowest replica in quorum)
Internet-scale Services

Most queries are simple, joins infrequent
- Look up price of item
- Add item to shopping cart
- Add like to comment

Conflicts are rare
- Many workloads are read- or write-heavy
- My cart doesn’t interfere with your cart

Scale out philosophy
- Use thousands of commodity servers
- Each table \textit{sharded} across hundreds to thousands of servers

Geographic replication
- Data centers across the world
- Tolerate failure of any one of them

Latency is key
- Documented financial impact of hundreds of \textit{milliseconds}
- Complex web pages made up of hundreds of queries

Consistency requirement can be relaxed
- Focus on availability and latency
~150 separate queries to render the home page
(Similar data in Facebook)
Focus on 99.9% latency

Each web page load has hundreds of objects
  ◦ Page load = latency of slowest object

Each user interacts with dozens of web pages
  ◦ Experience colored by slowest page

99.9% latency can be orders of magnitude higher

![Graph showing latency over time](image)
The Key-value Abstraction

(Business) Key $\rightarrow$ Value
(twitter.com) tweet id $\rightarrow$ information about tweet
(amazon.com) item number $\rightarrow$ information about it
(kayak.com) Flight number $\rightarrow$ information about flight, e.g., availability
(yourbank.com) Account number $\rightarrow$ information about it
The Key-value Abstraction (2)

It’s a dictionary datastructure.
- Insert, lookup, and delete by key
- E.g., hash table, binary tree

But distributed.

Sound familiar? Remember Distributed Hash tables (DHT) in P2P systems?

It’s not surprising that key-value stores reuse many techniques from DHTs.
Key-value/NoSQL Data Model

NoSQL = “Not Only SQL”

Necessary API operations: \texttt{get(key)} and \texttt{put(key, value)}
\begin{itemize}
  \item And some extended operations, e.g., “CQL” in Cassandra key-value store
\end{itemize}

Tables
\begin{itemize}
  \item “Column families” in Cassandra, “Table” in HBase, “Collection” in MongoDB
  \item Like RDBMS tables, but ...
  \item May be unstructured: May not have schemas
    \begin{itemize}
      \item Some columns may be missing from some rows
    \end{itemize}
  \item Don’t always support joins or have foreign keys
  \item Can have index tables, just like RDBMSs
\end{itemize}
Key-value/NoSQL Data Model

Unstructured

Columns Missing from some Rows

No schema imposed

No foreign keys, joins may not be supported

### users table

<table>
<thead>
<tr>
<th>user_id</th>
<th>name</th>
<th>zipcode</th>
<th>blog_url</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Alice</td>
<td>12345</td>
<td>alice.net</td>
</tr>
<tr>
<td>422</td>
<td>Charlie</td>
<td></td>
<td>charlie.com</td>
</tr>
<tr>
<td>555</td>
<td></td>
<td>99910</td>
<td>bob.blogspot.com</td>
</tr>
</tbody>
</table>

### blog table

<table>
<thead>
<tr>
<th>id</th>
<th>url</th>
<th>last_updated</th>
<th>num_posts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alice.net</td>
<td>5/2/14</td>
<td>332</td>
</tr>
<tr>
<td>2</td>
<td>bob.blogspot.com</td>
<td>10003</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>charlie.com</td>
<td>6/15/14</td>
<td></td>
</tr>
</tbody>
</table>
Column-Oriented Storage

NoSQL systems often use column-oriented storage.

RDBMSs store an entire row together (on disk or at a server).

NoSQL systems typically store a column together (or a group of columns).
- Entries within a column are indexed and easy to locate, given a key (and vice-versa).

Why useful?
- Range searches within a column are fast since you don’t need to fetch the entire database.
- E.g., Get me all the blog_ids from the blog table that were updated within the past month.
  - Search in the the last_updated column, fetch corresponding blog_id column.
  - Don’t need to fetch the other columns.
Next

Design of a real key-value store, 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

(Version from 2015)
Let’s go Inside Cassandra:
Key -> Server Mapping

How do you decide which server(s) a key-value resides on?
Cassandra uses a Ring-based DHT but without finger tables or routing. 

Key \rightarrow server mapping is the “Partitioner”

Say \( m = 7 \)
Data Placement Strategies

Replication Strategy: two options:
1. SimpleStrategy
2. NetworkTopologyStrategy

1. SimpleStrategy: uses the Partitioner, of which there are two kinds
   1. RandomPartitioner: Chord-like hash partitioning
   2. ByteOrderedPartitioner: Assigns ranges of keys to servers.
      ◦ Easier for range queries (e.g., Get me all twitter users starting with [a-b])

   ◦ Two replicas per DC
   ◦ Three replicas per DC
   ◦ Per DC
      ◦ First replica placed according to Partitioner
      ◦ Then go clockwise around ring until you hit a different rack
Snitches

Maps: IPs to racks and DCs. Configured in cassandra.yaml config file

Some options:

- **SimpleSnitch**: Unaware of Topology (Rack-unaware)
- **RackInferring**: Assumes topology of network by octet of server’s IP address
  - 101.201.202.203 = x.<DC octet>.<rack octet>.<node octet>
- **PropertyFileSnitch**: uses a config file
- **EC2Snitch**: uses EC2.
  - EC2 Region = DC
  - Availability zone = rack

Other snitch options available
Virtual Nodes

Randomized key placement results in imbalances
- Remember homework?

Nodes can be heterogeneous

Virtual nodes: each node has *multiple identifiers*
- $H(\text{node IP}||1) = 117$
- $H(\text{node IP}||2) = 12$

Node acts as both 117 and 12
- Stores two ranges, but each range is smaller (and more balanced)

Higher capacity nodes can have more identifiers
 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
- Per-key Coordinator ensures writes for the key are serialized

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? We’ll see later.
Writes (2)

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).

One ring per datacenter
- Per-DC coordinator elected to coordinate with other DCs
- Election done via Zookeeper, which runs a Paxos (consensus) variant
  - (Like Raft, but Greekier)
**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
   - Typically append-only datastructure (fast)
   - Cache that can be searched by key
   - Write-back cache as opposed to write-through

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
- **SSTables are immutable** (once created, they don’t change)
- Index file: An SSTable of (key, position in data sstable) pairs
- And a Bloom filter (for efficient search) – next slide
Bloom Filter

Compact way of representing a set of items
Checking for existence in set is cheap
Some probability of false positives: an item not in set may check true as being in set
Never false negatives

On insert, set all hashed bits.
On check-if-present, return true if all hashed bits set.
• False positives
False positive rate low: $m=4$ hash functions
100 items, 3200 bits
FP rate = 0.02%
Compaction

Data updates accumulate over time and SSTables and logs need to be compacted
  ◦ The process of compaction merges SSTables, i.e., by merging updates for a key
  ◦ Run periodically and locally at each server
Deletes

Delete: don’t delete item right away
- Add a tombstone to the log
- Eventually, when compaction encounters tombstone it will delete item
Reads

Read: Similar to writes, except

- Coordinator can contact X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X
  - (X? We’ll see later.)

- Coordinator also fetches value from other replicas
  - Checks consistency in the background, initiating a read repair if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date

- At a replica
  - Read looks at Memtables first, and then SSTables
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)
Membership

Any server in cluster could be the coordinator

So every server needs to maintain a list of all the other servers that are currently in the server

List needs to be updated automatically as servers join, leave, and fail
Cassandra uses gossip-based cluster membership

<table>
<thead>
<tr>
<th>Address</th>
<th>Heartbeat Counter</th>
<th>Time (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10120</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>10103</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>10098</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>10111</td>
<td>65</td>
</tr>
</tbody>
</table>

Protocol:
- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than $T_{fail}$, node is marked as failed

Current time: 70 at node 2
(asynchronous clocks)
Suspicion Mechanisms in Cassandra

Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior

Accrual detector: Failure Detector outputs a value (PHI) representing suspicion

Apps set an appropriate threshold

PHI calculation for a member
- Inter-arrival times for gossip messages
- PHI(t) = \(-\log(CDF \text{ or Probability}(t_{\text{now}} - t_{\text{last}}))/\log 10\)
- PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats

In practice, PHI = 5 => 10-15 sec detection time
Cassandra Vs. RDBMS

MySQL is one of the most popular (and has been for a while)

On > 50 GB data

MySQL
- Writes 300 ms avg
- Reads 350 ms avg

Cassandra
- Writes 0.12 ms avg
- Reads 15 ms avg

Orders of magnitude faster

What’s the catch? What did we lose?
Mystery of “X”: CAP Theorem

Proposed by Eric Brewer (Berkeley)

Subsequently proved by Gilbert and Lynch (NUS and MIT)

In a distributed system you can satisfy at most 2 out of the 3 guarantees:

1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
2. **Availability**: the system allows operations all the time, and operations return quickly
3. **Partition-tolerance**: the system continues to work in spite of network partitions
Why is Availability Important?

Availability = Reads/writes complete reliably and quickly.

Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.

At Amazon, each added millisecond of latency implies a $6M yearly loss.

User cognitive drift: If more than a second elapses between clicking and material appearing, the user’s mind is already somewhere else

SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.
Why is Consistency Important?

- Consistency = all nodes see same data at any time, or reads return latest written value by any client.

When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.

When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.
Why is Partition-Tolerance Important?

- Partitions can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- Still desire system to continue functioning normally under this scenario
Since partition-tolerance is essential in today’s cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability.

**Cassandra**
- Eventual (weak) consistency, Availability, Partition-tolerance

**Traditional RDBMSs**
- Strong consistency over availability under a partition
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.

- **Consistency**: Cassandra, RIAK, Dynamo, Voldemort
- **Partition-tolerance**: HBase, HyperTable, BigTable, Spanner
- **Availability**: RDBMSs (non-replicated)
Eventual Consistency

If all writes stop (to a key), then all its values (replicas) will converge eventually.

If writes continue, then system always tries to keep converging.
- Moving “wave” of updated values lagging behind the latest values sent by clients, but always trying to catch up.

May still return stale values to clients (e.g., if many back-to-back writes).

But works well when there a few periods of low writes – system converges quickly.
RDBMS vs. Key-value stores

While RDBMS provide **ACID**
- Atomicity
- Consistency
- Isolation
- Durability

Key-value stores like Cassandra provide **BASE**
- Basically Available Soft-state Eventual Consistency
- Prefers Availability over Consistency
Back to Cassandra: Mystery of X

Cassandra has **consistency levels**

Client is allowed to choose a consistency level for each operation (read/write)

- ANY: any server (may not be replica)
  - Fastest: coordinator caches write and replies quickly to client
- ALL: all replicas
  - Ensures strong consistency, but slowest
- ONE: at least one replica
  - Faster than ALL, but cannot tolerate a failure
- QUORUM: quorum across all replicas in all datacenters (DCs)
  - What?
Quorums?

In a nutshell:

Quorum = majority
  > 50%

Any two quorums intersect
  Client 1 does a write in red quorum
  Then client 2 does read in blue quorum

At least one server in blue quorum returns latest write

Quorums faster than ALL, but still ensure strong consistency

Five replicas of a key-value pair
Quorums in Detail

Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.

Reads
- Client specifies value of $R$ ($\leq N =$ total number of replicas of that key).
- $R =$ read consistency level.
- Coordinator waits for $R$ replicas to respond before sending result to client.
- In background, coordinator checks for consistency of remaining $(N-R)$ replicas, and initiates read repair if needed.
Quorums in Detail (Contd.)

Writes come in two flavors

- Client specifies $W \leq N$
- $W =$ write consistency level.
- Client writes new value to $W$ replicas and returns. Two flavors:
  - Coordinator blocks until quorum is reached.
  - Asynchronous: Just write and return.
Quorums in Detail (Contd.)

R = read replica count, W = write replica count

Two necessary conditions for consistency:
1. \( W + R > N \)
2. \( W > N/2 \)

Select values based on application
- \( (W=1, R=1) \): very few writes and reads
- \( (W=N, R=1) \): great for read-heavy workloads
- \( (W=N/2+1, R=N/2+1) \): great for write-heavy workloads
- \( (W=1, R=N) \): great for write-heavy workloads with mostly one client writing per key
Cassandra Consistency Levels (Contd.)

Client is allowed to choose a consistency level for each operation (read/write)
- ANY: any server (may not be replica)
  - Fastest: coordinator may cache write and reply quickly to client
- ALL: all replicas
  - Slowest, but ensures strong consistency
- ONE: at least one replica
  - Faster than ALL, and ensures durability without failures
- QUORUM: quorum across all replicas in all datacenters (DCs)
  - Global consistency, but still fast
- LOCAL_QUORUM: quorum in coordinator’s DC
  - Faster: only waits for quorum in first DC client contacts
- EACH_QUORUM: quorum in every DC
  - Lets each DC do its own quorum: supports hierarchical replies
Vector-clock Consistency (Dynamo)

How to reconcile?

Application
- E.g., add-to-cart: merge additions
- E.g., book flight: overbooking
- E.g., commit code change: merge conflict

System
- Last write wins

![Diagram](image)

Figure 3: Version evolution of an object over time.