CS190 (CS122D): Beyond SQL Data Management
—Lecture #5—

Mike Carey
UC Irvine
mjcarey@ics.uci.edu

M. Carey, Spring 2021: CS122D
Announcements

• Hopefully HW 1 is going smoothly by now!
  • Turn it in as per the TA instructions on Piazza...
  • Be sure to keep watching the Piazza page
    • Good use of Piazza so far...! (It’s how we hike the Rockies...)
  • I’m going to try and continue little weekly quizzes
    • Goal is to nudge you gently to keep up (w/low point value)
    • No partial credit (due to an auto-grading requirement)

• Today: Key-Value Stores
  • The first stop on our NoSQL systems tour!
Evolution of DBMS

Files

- Manual Coding
- Byte streams
- Majority of application development effort goes towards building and then maintaining data access logic

CODASYL/IMS

- Early DBMS Technologies
  - Records and pointers
  - Large, carefully tuned data access programs that have dependencies on physical access paths, indexes, etc.

Relational

- Relational DB Systems
  - Declarative approach
  - Tables + views bring “data independence”
  - Details left to system
  - Designed to simplify data-centric application development

M. Carey, Spring 2021: CS122D
NoSQL Data Store Features*

1. Horizontally scales “simple operation” throughput over many servers
2. Replicates and distributes (partitions) data over many servers
3. Provides a simple CLI or protocol (vs. SQL)
4. Has a weaker concurrency model than ACID SQL
5. Efficiently uses distributed indexes and RAM
6. Permits dynamically adding attributes to data records

(* R. Cattell, Scalable SQL and NoSQL Data Stores, 2011.)
Types of NoSQL Data Stores

![Diagram showing types of NoSQL data stores: Key Value, Column Family, Document, Graph, with Typical RDBMS at the bottom.](Image)
Key-Value Store Origins

**About Memcached** (a popular example)

memcached is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load.

You can think of it as a short-term memory for your applications.
Key-Value Store Origins

**About Memcached** *(a popular example)*
memcached is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load.

You can think of it as a short-term memory for your applications.

*(Can independently scale the database and caching tiers.)*
Basic Key-Value Store API

**put**(key, data)
- Store and associate data with key

**data** = **get**(key)
- Retrieve the data associated with key

**delete**(key)
- Delete the data associated with key

 Typically key is a string and data is a “black box” (binary or string). 2\textsuperscript{nd}ary indexes are not supported. (Adding new attributes within the data value is thus a non-problem.)
Can support multiple collections (“buckets”) to help with organizing the data:

**put**(key, data, bucket)
• Store and associate *data* with *key* in *bucket*

**data** = **get**(key, bucket)
• Retrieve the *data* associated with *key* in *bucket*

**delete**(key, bucket)
• Delete the *data* associated with *key* in *bucket***
Typical Key-Value Store Use Case

An shopping cart is an example of an aggregate object...
Key-Value Store Use Case (cont.)

mikejcarey@aol.com

Shopping Cart:

* Your cart is empty...

(timeouts, persistence, failures, ...)

M. Carey, Spring 2021: CS122D
One Example:  

- Open-source, in-memory “data structure store”
- Usable as a database, cache, or message broker
- Associates atomic **data structures** with keys:
  - Strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, geospatial indexes, streams, ...
- Built-in replication, Lua scripting, and transactions
- LRU eviction, different levels of on-disk persistence
- Provides HA (high availability) via Redis Sentinel and automatic partitioning (sharding) via Redis Cluster
- Brief Redis tutorials: [https://youtu.be/qr4FVhBTq0I](https://youtu.be/qr4FVhBTq0I) and [http://try.redis.io/](http://try.redis.io/).
Scaling Key-Value Stores

Hash or range (or hash-range) partitioning (a.k.a. “sharding”)

Logical bucket

Hash-based or range-based local storage
LSM-Based Local Storage  (Popular)

Periodically merge disk trees

Sequential writes to disk

Memory

Disk

Thanks to Dr. Sattam Alsubaiee (now Director for Insights, National Information Center – Saudi Arabia)
Data Distribution Models

- Single Server
- Master-Slave Replication
- Sharding
- Master-Slave Replication + Sharding
- Peer-to-Peer Replication
Single Server

Client Application

Driver

writes

reads

Primary
Master-Slave Replication
Master-Slave Client Access

Things to Note:

• All writes go to Master copy (Primary)
• Likewise for consistent reads
• Can use Slave (Secondary) copies to scale reads for which consistency is negotiable
More Things to Note:

- A critical detail is the ordering among the following events:
  - Primary’s write to disk
  - Primary’s writes to secondaries
  - Secondaries’ writes to disk
  - Operation acknowledgements (synchronous vs. asynchronous)

- This will affect:
  - What happens if a node fails
  - What happens if the network partitions
  - What happens when a node recovers
Master-Slave Failure Handling

Election for New Primary

Secondary \(\xrightarrow{\text{Heartbeat}}\) Secondary

New Primary Elected

Primary \(\xrightarrow{\text{Replication}}\) Secondary

Primary \(\xrightarrow{\text{Heartbeat}}\) Secondary
Master-Slave + Sharding

- All 3 nodes now handle 1/3 of the primary workload (in general, each of N nodes handles 1/N of the load)
- No longer need slaves to offload master read operations (depending on where the shards are geo-located, that is)
- Client application driver may be partition-aware

(“Chained declustering”)
Peer-to-Peer Replication

(“Multi-master” operation makes consistency “interesting” 😊)
So What About Consistency...?

• Challenges include
  • Applications needing to read and/or update multiple data items
  • Multiple clients concurrently accessing shared data
  • Potential system failures (storage, nodes, network, ...)

• Logical Consistency
  • Ensuring that different data items make sense together

• Replication consistency
  • Ensuring that different copies of a data item don’t diverge from one another (at least not long-term)
The **ACID** Properties (RDBMS)

- **Atomicity**: Each transaction is **all or nothing**
  - No worries about partial effects (if failures) and cleanup

- **Consistency**: Each transaction moves the database from one **consistent state** to another one
  - This is largely the application builder’s responsibility
  - Note that this is about *logical consistency*

- **Isolation**: Each transaction can be written as if it’s the **only transaction** in existence (**if so desired**)
  - Minimize concurrency worries when building applications

- **Durability**: Once a transaction has committed, its **effects will not be lost**
  - Application code doesn’t have to worry about data loss
Transactions in SQL

• A **transaction** is automatically started whenever a SQL statement accesses or modifies the database
  • SELECT, UPDATE, CREATE TABLE, INSERT, ...
  • Multi-statement transactions are supported

• A multi-statement transaction can be ended using
  • A **COMMIT** statement
  • A **ROLLBACK** statement (SQL-speak for **abort**)

• Each transaction runs under a combination of an **access mode** and an **isolation level**
Consistency Options in SQL

• The access mode controls what a transaction can potentially do to the database:
  • READ ONLY: not permitted to modify the DB
  • READ WRITE (default): allowed to modify the DB

• The isolation level controls a transaction’s exposure to other (concurrent) transactions:
  • READ UNCOMMITTED – can read “dirty data”
  • READ COMMITTED – cannot read dirty data
  • REPEATABLE READ – two reads will yield the same result
  • SERIALIZABLE – results as if transactions are run serially
Mechanisms & Scalability (RDBMS)

• **Concurrency control**
  • Two-phase locking (2PL) with read- and write-locks
  • Optimistic concurrency (commit-time validation check)
  • Possible versioning (MVCC)

• **Crash recovery**
  • Write-ahead logging (WAL protocol)
  • Transaction committed iff COMMIT record is on disk

• Hard to *scale* in a clustered setting (for large N)
  • Distributed locking (deadlocks), commit-time validation
  • Two-phase commit protocol (and its availability impact) for transactions that affect data on multiple nodes
Consistency Options in NoSQL

• For logical consistency, the first answer has been to support only *single-object* transactions
  • Avoids challenges of coordination at scale (large N)

• For replication consistency, the first answer has been to adopt a model based on *eventual consistency*
  • Replicas converge to the same set of values if the system is allowed to settle
  • Can avoid or identify divergence (or its visibility) by associating timestamps or timestamp vectors with data objects and using them appropriately when deciding which value(s) to believe (i.e., to order the versions)

• Some inconsistency is unavoidable / accepted in the world
  • Rain might have damaged the goods in a warehouse
  • Some overbooking of seats/rooms may be acceptable
Theorem (for Replicated Data)

• Choose two of these three desirable properties:
  • Consistency: Data is logically consistent and current
  • Availability: If you can talk to a node, it can R+W data
  • Partition Tolerance: Cluster can handle communication breakages (“split brain” syndrome)

• Essentially impossible to avoid network partitions, so in practice the design choices are CP or AP (not CA)
  • AP: Prioritize availability over consistency – e.g., allow possibly conflicting updates, and answer queries when partitioned
  • CP: Prioritize consistency over availability – e.g., disallow conflicting updates, don’t answer queries when partitioned

• Also see Abadi’s PACELC (“pass-elk”) amendment to CAP:
  • “If there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?”
Quorums & Versions

- Assume a cluster of $N$ nodes configured with **$K$-way replication** for each data partition
  - Safely writing to $W > K/2$ nodes will ensure that two conflicting writes have at least one node in common
  - Reading from $R > K-W$ nodes ensures that a given read has at least one node in common with the most recent write

For an object in a replicated partition, can track the number of updates that the object has gotten from each replica (node) as its "vector stamp"

Ex: $[n1: 1, n2: 2, n3: 5]$ vs. $[n1: 1, n2: 1, n3: 5]$, where the latter is missing an update from node n2, so the first version is newer (and would be returned as the latest)

A case like $[n1: 1, n2: 2, n3: 5]$ vs. $[n1: 2, n2: 1, n3: 5]$ is evidence of a write-write conflict that would need resolution

Vector needed for peer-to-peer replication – a single counter or Lamport timestamp

$K = 5$ copies
$W = 3$ nodes
$R = 3$ nodes
Quorums & Versions (cont.)

• Assume a cluster of N nodes configured with K-way replication for each data partition
  • Safely writing to $W > K/2$ nodes will ensure that two conflicting writes have at least one node in common
  • Reading from $R > K-W$ nodes ensures that a given read has at least one node in common with the most recent write

• For an object in a replicated partition, can track the number of updates that the object has gotten from each replica (node) as its “vector stamp”
  • Ex: $[n1: 1, n2: 2, n3: 5]$ vs. $[n1: 1, n2: 1, n3: 5]$, where the latter is missing an update from node n2, so the first version is newer (and would be returned as the latest for a read)
  • A case like $[n1: 1, n2: 2, n3: 5]$ vs. $[n1: 2, n2: 1, n3: 5]$ is evidence of a write-write conflict that requires resolution
  • Vectors needed for peer-to-peer replication – a single counter or timestamp would suffice for master-slave replication
To Be Continued....