Data-Intensive Distributed Computing
CS 431/631 451/651 (Fall 2019)

Part 7: Mutable State (1/2)
November 12, 2019

Ali Abedi
Structure of the Course

“Core” framework features and algorithm design

- Analyzing Text
- Analyzing Graphs
- Analyzing Relational Data
- Data Mining
The Fundamental Problem

We want to keep track of \textit{mutable} state in a \textit{scalable} manner

Assumptions:

State organized in terms of logical records
State unlikely to fit on single machine, must be distributed

MapReduce won’t do!

Want more? Take a \textit{real} distributed systems course!
The Fundamental Problem

We want to keep track of *mutable* state in a *scalable* manner

Assumptions:
- State organized in terms of logical records
- State unlikely to fit on single machine, must be distributed

*Uh... just use an RDBMS?*
What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support
RDBMSes: Pain Points
#1: Must design up front, painful to evolve
#2: Pay for ACID!
What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

What if we want *a la carte*?

Source: www.flickr.com/photos/vidiot/18556565/
Features *a la carte*?

What if I’m willing to give up consistency for scalability?
What if I’m willing to give up the relational model for flexibility?
What if I just want a cheaper solution?

Enter... NoSQL!
HOW TO WRITE A CV

DO YOU HAVE ANY EXPERTISE IN SQL?

NO

DOESN'T MATTER. WRITE: "EXPERT IN NO SQL”

Leverage the NoSQL boom
NoSQL
(Not only SQL)

1. Horizontally scale “simple operations”
2. Replicate/distribute data over many servers
3. Simple call interface
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schemas

But, don’t blindly follow the hype... Often, MySQL is what you really need!

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. SIGMOD Record.
“web scale”
(Major) Types of NoSQL databases

Key-value stores

Column-oriented databases

Document stores

Graph databases
Three Core Ideas

Partitioning (sharding)
To increase scalability and to decrease latency

Caching
To reduce latency

Replication
To increase robustness (availability) and to increase throughput
Key-Value Stores
Key-Value Stores: Data Model

Stores associations between keys and values

Keys are usually primitives
For example, ints, strings, raw bytes, etc.

Values can be primitive or complex: often opaque to store
Primitives: ints, strings, etc.
Complex: JSON, HTML fragments, etc.
Key-Value Stores: Operations

Very simple API:
Get – fetch value associated with key
Put – set value associated with key

Optional operations:
- Multi-get
- Multi-put
- Range queries
- Secondary index lookups

Consistency model:
- Atomic single-record operations (usually)
- Cross-key operations: who knows?
Key-Value Stores: Implementation

Non-persistent:
Just a big in-memory hash table
Examples: Redis, memcached

Persistent
Wrapper around a traditional RDBMS
Examples: Voldemort

What if data doesn’t fit on a single machine?
Simple Solution: Partition!

Partition the key space across multiple machines
Let’s say, hash partitioning
For n machines, store key $k$ at machine $h(k) \ mod \ n$

Okay... But:
How do we know which physical machine to contact?
How do we add a new machine to the cluster?
What happens if a machine fails?
Clever Solution

Hash the keys
Hash the machines also!

Distributed hash tables!
(following combines ideas from several sources...)

22
$h = 2^n - 1 \quad h = 0$
Routing: Which machine holds the key?

Each machine holds pointers to predecessor and successor

Send request to any node, gets routed to correct one in $O(n)$ hops

Can we do better?
Routing: Which machine holds the key?

Each machine holds pointers to predecessor and successor

$+ \text{“finger table” (+2, +4, +8, ...)}$

$h = 2^n - 1 \quad h = 0$

Send request to any node, gets routed to correct one in $O(\log n)$ hops
Routing: Which machine holds the key?
New machine joins: What happens?

How do we rebuild the predecessor, successor, finger tables?

$h = 2^n - 1 \quad h = 0$

Stoica et al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM.

Cf. Gossip Protocols
Machine fails: What happens?

Solution: Replication

Covered!

$h = 2^n - 1$

$h = 0$
Three Core Ideas

Partitioning (sharding)
To increase scalability and to decrease latency

Consistency?
Replication
To increase robustness (availability) and to increase throughput

Keeping track of the partitions?

Caching
To reduce latency
Another Refinement: Virtual Nodes

Don’t directly hash servers

Create a large number of virtual nodes, map to physical servers
  Better load redistribution in event of machine failure
  When new server joins, evenly shed load from other servers
Bigtable
Bigtable Applications

Gmail
Google’s web crawl
Google Earth
Google Analytics
Data source and data sink for MapReduce

HBase is the open-source implementation...
Data Model

A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map

Map indexed by a row key, column key, and a timestamp
(row:string, column:string, time:int64) → uninterpreted byte array

Supports lookups, inserts, deletes
Single row transactions only

Image Source: Chang et al., OSDI 2006
Rows and Columns

Rows maintained in sorted lexicographic order
Applications can exploit this property for efficient row scans
Row ranges dynamically partitioned into tablets

Columns grouped into column families
Column key = family:qualifier
Column families provide locality hints
Unbounded number of columns

At the end of the day, it’s all key-value pairs!
### Key-Values

<table>
<thead>
<tr>
<th>row, column family, column qualifier, timestamp</th>
<th>value</th>
</tr>
</thead>
</table>


Okay, so how do we build it?

<table>
<thead>
<tr>
<th>In Memory</th>
<th>On Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutability Easy</td>
<td>Mutability Hard</td>
</tr>
<tr>
<td>Small</td>
<td>Big</td>
</tr>
</tbody>
</table>
Log Structured Merge Trees

What happens when we run out of memory?
Log Structured Merge Trees

- Writes to MemStore
- MemStore to Reads
- Flush to disk
- Store

Immutable, indexed, persistent, key-value pairs

What happens to the read path?
Log Structured Merge Trees

What happens as more writes happen?
Log Structured Merge Trees

What happens to the read path?
Log Structured Merge Trees

MemStore

Write

Memory

Disk

Flush to disk

Merge

Reads

Store

Store

Store

Store

Immutable, indexed, persistent, key-value pairs

What’s the next issue?
Log Structured Merge Trees

**Writes** → **MemStore** → **Merge** → **Reads**

Memory → **Flush to disk** → **Store** → **Store** → **Store**

Immutable, indexed, persistent, key-value pairs

Compaction!
Log Structured Merge Trees

- **Writes** to **MemStore**
- **Merge** operation
- **Flush to disk**
- **Reads** from **Store**

**Store** is immutable, indexed, persistent, key-value pairs.
Log Structured Merge Trees

Writes → MemStore → Reads

Memory → WAL

Disk → Store

Immutable, indexed, persistent, key-value pairs

One final component...
Log Structured Merge Trees
The complete picture...

Writes → MemStore → Merge → Reads

Memory
Disk

Flush to disk

Logging for persistence

WAL

Store
Store
Store

Immutable, indexed, persistent, key-value pairs

Compaction!
Log Structured Merge Trees

The complete picture...

Okay, now how do we build a distributed version?
HBase: Bigtable building blocks

- HDFS
- GFS
- HFile
- SSTable
- Tablet
- Region
- Tablet Server
- Regions Server
- Chubby
- Zookeeper
SSTable

Persistent, ordered immutable map from keys to values

Stored in GFS: replication “for free”

Supported operations:

Look up value associated with key
Iterate key/value pairs within a key range
Region

Tablet

Dynamically partitioned range of rows
Comprised of multiple SSTables
Region Server

Tablet Server

Writes → MemStore → Reads

Memory

Disk

Logging for persistence

Flush to disk

WAL

SSTable

SSTable

SSTable

Immutable, indexed, persistent, key-value pairs

Compaction!
Table

Comprised of multiple tablets
SSTables can be shared between tablets
Each tablet is assigned to one tablet server at a time

Exclusively handles read and write requests to that tablet

What happens when a tablet grows too big?
What happens when a tablet server fails?

We need a lock service!
HBase: Bigtable building blocks

- HDFS
- GFS
- HFile
- SSTable
- Tablet
- Region
- Regions Server
- Tablet Server
- Chubby
- Zookeeper
Architecture

Client library

Bigtable master **HMaster**

Tablet servers

**Regions Servers**
Bigtable Master

Roles and responsibilities:
- Assigns tablets to tablet servers
- Detects addition and removal of tablet servers
  - Balances tablet server load
  - Handles garbage collection
  - Handles schema changes

Tablet structure changes:
- Table creation/deletion (master initiated)
- Tablet merging (master initiated)
- Tablet splitting (tablet server initiated)
Compactions

Minor compaction
Converting the memtable into an SSTable
Reduces memory usage and log traffic on restart

Merging compaction
Reads a few SSTables and the memtable, and writes out a new SSTable
Reduces number of SSTables

Major compaction
Merging compaction that results in only one SSTable
No deletion records, only live data
Table

Comprised of multiple tables
SSTables can be shared between tablets

Tablet
aardvark - base

Tablet
basic - database

SSTable
SSTable
SSTable
SSTable
SSTable
SSTable

How does this happen?
Three Core Ideas

**Partitioning (sharding)**
To increase scalability and to decrease latency

**Replication**
To increase robustness (availability) and to increase throughput

**Caching**
To reduce latency

*Keeping track of the partitions?*

*Consistency?*