Today’s topics

- No SQL storage
Using quorum-like system

- **R**
  - Read Quorum
  - Minimum number of nodes that must participate in a successful read operation
- **W**
  - Write Quorum
  - Minimum number of nodes that must participate in a successful write operation

- Setting **R** and **W** for the given replication factor of **N**
  - **R** + **W** > **N**
  - **W** > **N**/2

- The latency of a get (or put) operation is dictated by the slowest one of the **R** (or **W**) replicas
  - **R** and **W** are configured to be less than **N**

**put** request

- Coordinator node

1. Generates the *vector clock*  
   --For the new version

2. Writes the new version locally

3. Sends the new version to the **N** highest-ranked reachable nodes  
   --Along with the new vector clock
**get request**

- The coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes
  - In the preference list

- **Waits for** $R$ **responses**

- If multiple versions of the data are collected
  - Returns all the versions it deems to be causally unrelated

- The reconciled version superseding the current versions is written back

---

**Part 2. Large scale data storage system**

**NoSQL Storage: Key-Value Stores (Dynamo)**

1. Partitioning
2. High Availability for writes
3. Handling temporary failures
4. Recovering from permanent failures
5. Membership and failure detection
Sloppy quorum

• All read and write operations are performed on the **first N healthy nodes** from the preference list
  • May not always be the first N nodes on the hashing ring

• **Hinted handoff**
  • If a node is temporarily unavailable, **data is propagated to the next node in the ring**
  • Metadata contains information about the originally intended node
  • Stored in a separate local database and scanned periodically

• Upon detecting that the original node is recovered,
  • A data delivery attempt will be made
  • Once the transfer succeeds, the data at the temporary node will be removed

Example: Updated

The data will be sent to the node D

If C is temporarily down

After the recovery, D will send data to C

Then, it will remove the data.
What if $W$ (write quorum) is 1?

• Applications that need the highest level of availability can set $W$ as 1
  • Under Amazon’s model

• A write is accepted as long as a single node in the system has durably written the key to its local store

• A write request is rejected,
  • Only if all nodes in the system are unavailable

Part 2. Large scale data storage system

NoSQL Storage: Key-Value Stores (Dynamo)

(1) Partitioning
(2) High Availability for writes
(3) Handling temporary failures
(4) Recovering from permanent failures
(5) Membership and failure detection
Anti-entropy protocol

- Replica synchronization protocol
- **Hinted replica** can be lost before they can be returned to the original replica node
- Detect inconsistencies between the replicas faster
- **Minimize the amount of transferred data**
- Dynamo uses **Merkle tree**

Merkle tree

- **Hash tree** where leaves are hashes of the values of individual keys
  - Parent nodes are hashes of their respective children
- Each branch of the tree can be checked independently
  - Without requiring nodes to download the entire tree or dataset
- **If the hash values of the root of two trees are equal**
  - The values of the leaf nodes in the tree are equal
  - No synchronization needed
Uses of Merkle tree

- Merkle trees can be used to verify any kind of data stored, handled and transferred.
  - Used in a peer-to-peer network

- Trusted computing systems
  - Sun’s ZFS (Zeta File System)

- Google’s Wave protocol
- Git
- Cassandra and Dynamo
- Bittorrent protocol

How Merkle tree works

1. Data 1
2. Data 2
3. Data 3
4. Data 4

- $H_1+H_2$ means concatenating two values
- Top Hash Hash value of $(H_5+H_6)$
- $H_5$ Hash value of $(H_1+H_2)$
- $H_6$ Hash value of $(H_3+H_4)$
How Dynamo uses Merkle tree

- Each node maintains a separate Merkle tree for each key range
- Two nodes exchange the root of the Merkle tree
  - Corresponding to the key ranges that they host in common
- Node performs tree traversal and determines if there is any difference
  - Perform the appropriate synchronization action
- Disadvantage
  - When a new node joins or leaves
    - Tree needs to be recalculated

How Merkle tree works for Dynamo

1. Compare Top hashes
2. If top hash did not match
   - Compare H5 vs. H13 and H6 vs. H14
3. If H6 and H14 did not match
   - Compare H3 vs. H12 and H4 vs. H13
Part 2. Large scale data storage system

NoSQL Storage: Key-Value Stores (Dynamo)

(1) Partitioning
(2) High Availability for writes
(3) Handling temporary failures
(4) Recovering from permanent failures
(5) Membership and failure detection

Identifier "Ring" Membership

• A node outage should not result in re-balancing of the partition assignment or repair of the unreachable replicas
  • A node outage is mostly temporary

• Gossip-based protocol
  • Propagates membership changes
  • Maintains an eventually consistent view of membership

• Each node contacts a peer every second
  • Random selection
  • Two nodes reconcile their persisted membership change history
Logical partitioning

- Almost concurrent addition of two new nodes
  - Node A joins the ring
  - Node B joins the ring

- A and B consider themselves members of the ring
  - Yet neither would be immediately aware of each other
  - A does not know the existence of B
  - Logical partitioning

External Discovery

- Addresses the logical partitioning

- Seeds
  - Discovered via an external mechanism
  - Known to all nodes
  - Statically configured (or from a configuration service)

- Seed nodes will eventually reconcile their membership with all of the nodes
Failure Detection

• Attempts to
  • Avoid communication with unreachable peers during a get or put operation
  • Transfer partitions and hinted replicas

• Detecting communication failures
  • When there is no response to an initiated communication

• Responding to communication failures
  • Sender will try alternate nodes that map to failed node’s partitions
  • Periodically retry failed node for recovery

Part 2. Large scale data storage system
NoSQL Storage: Column Family Stores
Google’s Big Table
This material is built based on,


Column-family storage

- Optimized for the data
  - Sparse columns and no schema

- Aggregate-oriented storage
  - Most data interaction is done with the same aggregate
  - Aggregate
    - A collection of data that we interact with as a unit

- Stores groups of columns (column families) together
Storing data in a column-family store

- The stores organize their columns into column families
- Each column may be part of a single column family
- The column acts as unit for access
  - The assumption is that data for a particular column family will be usually accessed together
BigTable

• Google’s first answer to the question
  • “How do you store semi-structured data at scale?”

Scalability and latency

• Scale in capacity
  • E.g., webtable
    • 100,000,000,000 pages * 10 versions per page * 20KB/version
    • 20PB of data (200 million gigabytes)
  • E.g., google maps
    • 100TB of satellite image data

• Scale in throughput
  • Hundreds of millions of users
  • Tens of thousands to millions of queries per second

• Low latency
  • A few dozen milliseconds of total budget “inside” Google
  • Probably have to involve several dozen internal services per request
  • Few milliseconds for lookup
BigTable has been used by,

- Web indexing
- Google Reader
- Google Maps
- Google Book Search
- Google Earth
- Blogger.com
- Google Code
- YouTube
- Gmail
- ...

BigTable

- Provides a simple data model
  - Dynamic control over the data layout and format
  - Allows clients to reason about the locality properties of the data represented in the underlying storage

- Data is indexed using row and column names that can be arbitrary strings

- Data in BigTable
  - Uninterpreted strings
  - Clients often serialize various forms of structured and semi-structured data into these strings
BigTable

- Clients can control locality of their data
- Clients can control whether to serve data out of memory or from disk

Topics in BigTable

1. Data model
2. Locating tablet
3. Data Compaction
4. Data Compression
5. Caching and prefetching
Part 2. Large scale data storage system

NoSQL Storage: Column Family Stores

Google’s Big Table

1. Data model
2. Locating tablet
3. Data Compaction
4. Data Compression
5. Caching and prefetching

Data Model

• A BigTable is a **sparse, distributed, persistent multi-dimensional sorted map**
• The map is indexed by,
  - A row key
  - A column key
  - A timestamp

• Each value in the map is an uninterpreted array of bytes

```plaintext
(row:string, column:string, time:int64) \rightarrow string
```
Example of data model with Webtable

- Webtable
  - A large collection of web pages and related information
    - URLs
    - Contents
    - Information

```
<html>...</html>
```

Rows

- **Row keys**
  - Arbitrary strings
  - Every read or write of data under a single row key is atomic

- BigTable maintains data in lexicographic order by row key

- Row range for a table
  - Dynamically partitioned
Tablets

• Large tables are broken into tablets at row boundaries
  • A tablet holds a contiguous range of rows
  • Clients can often choose row keys to achieve locality
  • Aim for ~ 100MB to 200MB of data per tablet

• Serving machine responsible for ~100 tablets
  • Fast recovery
    • Allows a 100 machines to each pick up 1 tablet from the failed machine
  • Fine-grained load balancing
    • Migrate tablets away from the overloaded machine
    • Master makes load-balancing decisions

Tablets

• Read of short row ranges are efficient
  • Require communication with only a small number of machines
  • Clients get good locality for their data access

• maps.google.com/index.html is stored using the key com.google.maps/index.html
  • Storing pages under the same domain near each other makes some host and domain analysis more efficient
Column Families [1/2]

- Column keys are grouped into sets called **column families**
  - **Basic unit of access control**

- All data stored in a column family is usually of **the same type**
  - BigTable **compresses** data in the same column family together

- A column family must be **created before data can be stored** under any column key in that family
  - After a family has been created, any column key within the family can be used

Column Families [2/2]

- **Column key**
  - *family:qualifier*
    - Family name must be printable
    - Qualifier may be an arbitrary string

- **Access control and disk/memory accounting**
  - Performed at the **column family level**
Example: **Webtable with multiple column-families**

![Webtable with multiple column-families diagram]

---

**Timestamps**

- Each cell in Bigtable can contains multiple versions of the same data
  - Indexed by timestamp

- **BigTable timestamp**
  - 64-bit integers
  - Assigned by BigTable
    - Real time in microseconds
    - Explicitly assigned by client application

- Application should generate unique timestamp to avoid collisions
- Different versions of a cell are stored in decreasing timestamp order
  - The most recent versions can be read first
API

• Functions for creating and deleting tables and column families
• Changing cluster, table, and column-family metadata (access control rights)

```cpp
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Garbage collection

• Two per-column-family settings
  • Tell Bigtable to garbage-collect cell versions automatically
  • The last \( n \) versions are kept
    • i.e. only recent versions are kept
Part 2. Large scale data storage system

NoSQL Storage: Column Family Stores

Google’s Big Table

1) Data model

2) Locating tablet

3) Data Compaction

4) Data Compression

5) Caching and prefetching

System Structure

BigTable master

Performs metadata ops + load balancing

BigTable tablet server

Serves data

BigTable client

BigTable client library

BigTable tablet server

Serves data

BigTable tablet server

Serves data

Cluster scheduling system

Handles failover, monitoring

GFS

Holds tablet data, logs

Lock service

Holds metadata, handles master-collection
Building blocks

12/2/2019 and 12/4/2019

CS435 Introduction to Big Data – Fall 2019

W14.A.46

(1/2)

• **Memtable**: in-memory table
  • writes goes to log then to in-memory table
  • Periodically data are moved from memory table to disk (using SSTable file format)

• The Google **SSTable (Sorted String Table)** file format
  • Internally used to store the contents of a part of table (Tablet)
  • Persistently ordered immutable map from key to values
  • Keys and values are arbitrary byte strings

• **Tablet**
  • All of the SSTables for one key range + memtable

Building blocks

(2/2)

• SSTable contains a **sequence of blocks**
  • 64KB, configurable

• **Block index**
  • Stored at the end of SSTable
  • Index is loaded into memory when the SSTable is opened

• SSTable is used by: Cassandra, Hbase, LevelDB
• Open-source implementation
  • http://code.google.com/p/leveldb/
**SSTable: Sorted String Table**

Reading and writing data can dominate running time
Random reads and writes are critical features

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Key</th>
<th>Value</th>
<th>Key</th>
<th>Value</th>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
</tr>
<tr>
<td>Key</td>
</tr>
<tr>
<td>Key</td>
</tr>
<tr>
<td>Key</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Access to the block

- In-memory map of keys to \{SSTables, memtable\}
- Lookup can be performed with a single disk seek
  - Find the block by performing a binary search of the in-memory index
  - Read the block from disk
Locating tablets [1/2]

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row

- Using the BigTable master
  - Central server almost certainly would be bottleneck in large system
  - Instead: store special tables containing tablet location info in BigTable cell itself

Locating tablets [2/2]

- **3-level** hierarchical lookup scheme for tablets
  - Location is `ip:port` of relevant server
  - 1st level: bootstrapped from Chubby (lock service), points to the *root tablet*
  - 2nd level: Uses root tablet data to find owner(node) of appropriate *metadata tablets*
  - 3rd level: metadata table holds locations of tablets of all other tables: Metadata tablet itself can be split into multiple tablets

[Diagram showing the 3-level hierarchical lookup scheme for tablets, with pointers and locations for various tables and metadata.]
• Root tablet is **never split**
  • To ensure that the tablet location hierarchy has no more than 3 levels

• **Metadata tablet**
  • Stores the **location of a tablet** under a **row key**
    • Tablet’s identifier and its end row
  • Each metadata row stores approximately 1KB of data in memory
    • Average limit of 128MB Metadata tablets
      • $2^{34}$ tablets are addressed

---

**Caching the tablet locations** [1/4]

• Client library **caches tablet locations**

• **Traverses up** the tablet location hierarchy
  • If the client does not know the location of a tablet
  • If it discovers that the cached location information is **incorrect**
Caching the tablet locations [2/4]

• If the client’s cache is empty?
  • One read from Chubby
  • One read from root tablet
  • One read from metadata tablet
  • Three network round-trips is required to locate the tablet

Caching the tablet locations [3/4]

• If the client’s cache is stale?
  • With given information, client could not find the data
  • What is the maximum round-trips needed (if the root server has not changed)?
Caching the tablet server locations

- If the client’s cache is stale? (location of root table, metadata table, and actual tablet server)
  - With given information, client could not find the data
- First round: user accesses tablet and misses data (arrow 1)
- If only the tablet information is staled
  - 2 additional rounds to locate tablet info from the metadata tables (a-1, a-2)
- If the location of the metadata table info is also staled
  - 4 additional rounds
    - To the metadata table (it misses tablet info due to the stale info) (b-1)
    - To the root server to retrieve the location of the metadata table (b-2)
    - To the metadata table to retrieve the tablet server location(b-3)
    - Locate tablet from the tablet server(b-4)

Prefetching tablet locations

- Client library reads the metadata for more than one tablet
  - Whenever it reads the metadata table
- No GFS accesses are required
  - Table locations are stored in memory
Tablet Assignment (1/2)

• Each tablet is assigned to one tablet server at a time
  • The master keeps track of:
    • The set of live tablet servers
    • Which tablets are assigned
  
• New tablet assignment
  • The master assigns the tablet by sending a tablet load request to the tablet server

Tablet Assignment (2/2)

• A tablet server starts
  • Chubby creates a uniquely-named file in a specific Chubby directory
  • Exclusive lock
  • Master monitors this directory to discover tablet servers

• A tablet server terminates
  • Release its lock
  • Master will reassign its tablets more quickly
Tablet status

- The persistent state of a tablet is stored in GFS

Tablet Representation

- Write buffer in memory (random-access)
- MemTable
- SSTable on GFS
- SSTable on GFS
- SSTable on GFS
- Append-only log on GFS

SSTable: immutable on-disk ordered map from string→string
String keys <row, column, timestamp> triples
**write** operation

- The tablet server checks,
  - If the data is well-formed
  - If the user is authorized to mutate data

- Operation is **committed to a log file**

- The contents are inserted into the **MemTable**

---

**read** operation

- Tablet server checks
  - If the request is well-formed
  - If the user is authorized to read data

- Merged view of **MemTable**(in memory) and **SSTable**(in disk)
  - Read operation is performed
Part 2. Large scale data storage system

NoSQL Storage: 2. Column Family Stores

Google’s Big Table

(1) Data model
(2) Locating tablet
(3) Data Compaction
(4) Data Compression
(5) Caching and prefetching

Data Compaction and Compression

• What is the difference between data compaction and data compression?
Minor Compactions

- **As write operations executed**
  - The size of the memtable increases

- **Minor compaction**
  - When the memtable size reaches a threshold
    - The memtable is frozen
    - A new memtable is created
    - A frozen memtable is converted to an SSTable (stored in GFS)
  - Shrinks the memory usage in the tablet server
  - Reduces the amount of data that has to be read from the commit log during recovery (if the server dies)

Merging Compaction

- **New SSTable from the minor compaction will increase**
  - Read operations need to merge updates from large number of SSTables

- **Merging Compaction**
  - Bounds the number of such files periodically
  - Reads the contents of a few SSTables and the memtable and writes out a new SSTable
  - Input SSTables and memtable can be discarded as soon as the merging compaction has finished
Major Compaction

- Rewrites multiple SSTables into exactly one SSTable
  - No deletion information or deleted data included

Part 2. Large scale data storage system
NoSQL Storage: 2. Column Family Stores
  Google’s Big Table
  (1) Data model
  (2) Locating tablet
  (3) Data Compaction: Log-Structured Merge (LSM) Trees
  (4) Data Compression
  (5) Caching and prefetching
Background

- Sequential access to disk (magnetic or SSD) is at least three orders of magnitude faster than random IO
  - Journaling, logging or a heap file is fully sequential
  - 200-300 MB/s per drive

- But transitional logs are only really applicable to “SIMPLE” workloads
  - Data is accessed entirely
  - Data is accessed by a known offset

Sequential IO vs. Random IO

Do we have sequential datasets in BigTable?
Existing approaches to improve performance

- Hash
- B+ tree
- External file: create separate hash or tree index

- Adding index structure improves read performance
  - It will slow down write performance
  - Update structure and index

- Log-structured merge trees
  - Fully disk-centric
  - Small memory footage
  - Improved write performance
  - Read performance is still slightly poorer than B+ tree
Basic idea of LSM trees

- LSM trees manage **batches** of writes to be saved
- Each file contains a batch of changes covering a short period of time
  - Each file is sorted before it is written
- Files are **immutable**
- **New updates will create new files**
- Reads inspect all files
- Periodically files are merged

In-memory buffer for LSM (**MemTable**)

- Data is stored as a tree (Red-Black, B-tree etc) to preserve key-ordering
  - MemTable is replicated on disk as a write-ahead-log
- When the MemTable fills the sorted data is flushed to a new file on disk
- **Only sequential IO is performed**
- Each file represents a small, chronological subset of changes (sorted)
- Periodically the system performs a **compaction**
Conceptual view of rolling merge

Locality groups

- Clients can group multiple column families together into a locality group
  - Separate SSTable is generated for each locality group in each tablet

- Example
  - Locality group 1: Page metadata in Webtable
    - Language and checksum
  - Locality group 2: Contents of the page

- Application reading the metadata does not need to read through all of the page content
Part 2. Large scale data storage system
NoSQL Storage: 2. Column Family Stores

Google’s Big Table

(1) Data model
(2) Locating tablet
(3) Data Compaction
(4) Data Compression
(5) Caching and prefetching

Compression

- **Compression** is required for the data stored in BigTable
  - Similar values in the same row/column
    - With different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows

- Clients can control whether or not the **SSTables for a locality group are compressed**
  - User specifies the locality group to be compressed and the compression scheme
  - **Keep blocks small for random access** (~64KB compressed data)
  - Low CPU cost for encoding/decoding
    - Server does not need to encode/decode entire table to access a portion of it
Two-pass compression scheme

• Data to be compressed
  • Keys in BigTable (row, column and timestamp)
  • Sorted strings
  • Values in BigTable
    • BMDiff (Bentley and McIlroy’s Scheme) across all values in one family
    • BMDiff output for values 1..N is dictionary for value N+1

• Zippy is used for final pass over whole block
  • Localized repetitions
  • Cross-column-family repetition, compresses keys

• First pass: BMDiff
• Second pass: Zippy (now called as snappy)

BMDiff

• Jon Bentley, Douglas McIlroy, “Data compression using long common strings” In Data Compression Conference (1999), pp. 287-295

• Adapted to VCDiff (RFC3284)
  • Shared Dictionary Compression over HTTP (SDCH)
  • Chrome browser

• http://tools.ietf.org/html/rfc3284
Example of the Constitution of the US and the King James Bible

<table>
<thead>
<tr>
<th>File</th>
<th>Text</th>
<th>gzip</th>
<th>Relative compressed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>49523</td>
<td>13936</td>
<td>1.0</td>
</tr>
<tr>
<td>Const+Const</td>
<td>99046</td>
<td>26631</td>
<td>1.911</td>
</tr>
<tr>
<td>Bible</td>
<td>4460056</td>
<td>1321495</td>
<td>1.0</td>
</tr>
<tr>
<td>Bible+Bible</td>
<td>8920112</td>
<td>2642389</td>
<td>1.9995</td>
</tr>
</tbody>
</table>

doi: 10.1109/DCC.1999.755678
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=755678&isnumber=16375

Text and blocks

- Recognizes the second occurrence of the input text as a repetition
  - The second string is represented with a reference to the first

- Finding long common strings
  - The compression block size \( b \)
    - Between 20 and 1000
    - Ignore repeated strings of length less than \( b \)
  - For the file with length \( n \)
    - \( n/b \) fingerprints will be stored

- Hash table
  - To find common fingerprints and locate common strings
The compression algorithm

- Representing the common string
  - `<start, length>`
    - start: initial position
    - length: size of the common sequence

- e.g. “the Constitution of the United States PREAMBLE We, the people of the United States, in order to form a more perfect Union, …”
- → “the Constitution of the United States PREAMBLE We, the people <16,21>, in order to form a more perfect Union, …”

Implementation of Data compression

Text

Calculate hash value

Lookup the hash table

Hash table of fingertable

If there is no match, this value is added
What if we find a match?

- \( b = 100 \)
- The current block of length \( b \) matches block 56
- We could encode that single block as
  - \(<5600, 100>\)
  - This scheme guarantees not to encode any common sequences less than \( b \)

---

### Results

<table>
<thead>
<tr>
<th>File</th>
<th>Text</th>
<th>gzip</th>
<th>Relative compressed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>49523</td>
<td>13936</td>
<td>1.0</td>
</tr>
<tr>
<td>Const+Const</td>
<td>99046</td>
<td>26631</td>
<td>1.911</td>
</tr>
<tr>
<td>Bible</td>
<td>4460056</td>
<td>1321495</td>
<td>1.0</td>
</tr>
<tr>
<td>Bible+Bible</td>
<td>8920112</td>
<td>2642389</td>
<td>1.9995</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compression</th>
<th>bible</th>
<th>bible+bible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>4460056</td>
<td>8920112</td>
</tr>
<tr>
<td>gzip</td>
<td>1321495</td>
<td>2642389</td>
</tr>
<tr>
<td>com50</td>
<td>4384403</td>
<td>4384414</td>
</tr>
<tr>
<td>com20</td>
<td>3906771</td>
<td>3906782</td>
</tr>
<tr>
<td>com50</td>
<td>gzip</td>
<td>1318687</td>
</tr>
<tr>
<td>com20</td>
<td>gzip</td>
<td>1362413</td>
</tr>
</tbody>
</table>
Snappy

- Based on LZ77
  - Dictionary coders
  - Sliding window

- Very fast and stable but not high compression ratio
  - 20~100% lower compression ratio than gzip

BigTable and data compressions

- Large window data compression
  - BMDiff (~ 100MB/s for write, ~1000MB/sec for read)
  - Identify large amounts of shared boilerplate in pages from same host

- Small window data compression
  - Looks for repetitions in 16KB window
  - Snappy

- e.g. 45.1TB of crawled dataset (2.1B pages)
  - 4.2 TB compressed size
Part 2. Large scale data storage system
NoSQL Storage: 2. Column Family Stores

Google’s Big Table

(1) Data model
(2) Locating tablet
(3) Data Compaction
(4) Data Compression
(5) Caching and prefetching

Caching for read performance

• Tablet servers use two levels of caching
• Scan cache
  • Higher-level cache
  • Caches the key-value pairs returned by the SSTable interface in the table server
• Block cache
  • Lower-level cache
  • Caches SSTables blocks that were read from GFS
Bloom filters

- Read operation has to read from all SSTables that make up the state of a tablet
  - SSTables in disk results many disk accesses

- Bloom filter
  - Detects if an SSTable might contain any data for a specified row/column pair

- Probabilistic data structure
  - Tests whether the element is a member of a set
  - The element either definitely is not in the set or may be in the set

PART 2. LARGE SCALE DATA STORAGE SYSTEMS
DATA EXCHANGE MODEL

Sangmi Lee Pallickara, (Guest Lecturer: Paahuni Khandelwal)
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435
FAQs

• Term project presentation
  • 12 minutes per team
    • Presentation
    • Q&A
    • Transition

• Your questions/comments/attendance will be tracked (Participation score 5/100)

• Submit your slides 2 hrs before the class starts via Canvas

Topics

• Data Exchange Model
  • RESTful service interface
Part 2. Large scale data storage system

Data Exchange Model

Wearable devices and sensors
Fitbit APIs

- Store, read, analyze user’s activity data
- Data collected from user’s devices are stored in anywhere available
- Immediate and historical analysis

For more information: https://dev.fitbit.com/build/reference/
Fitbit APIs

- **Device API**
  - Accelerometer, Barometer, Clock, Console, Display, Heartrate, etc.
- **Settings API**
  - Creates application configuration
- **Companion API**
  - For applications running within the Fitbit mobile applications
  - Crypto, file-transfer, geolocation, storage, location-change, etc.
- **Web API**
  - Accesses information collected by trackers

Example: Activity & Exercise Logs

```
GET https://api.fitbit.com/1/user/[user-id]/activities/date/[date].json
```

<table>
<thead>
<tr>
<th>user-id</th>
<th>The encoded ID of the user. Use &quot;.&quot; (dash) for current logged-in user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The date in the format yyyy-MM-dd</td>
</tr>
<tr>
<td>Accept-Locale</td>
<td>optional</td>
</tr>
<tr>
<td>Accept-Language</td>
<td>optional</td>
</tr>
</tbody>
</table>
Example: Activity & Exercise Logs: Response

```
{
  "activities": [  
    {  
      "activityId":51007,
      "activityParentId":90019,
      "calories":230,
      "description":7mph",
      "distance":2.04,
      "duration":1097053,
      "hasStartTime":true,
      "isFavorite":true,
      "logId":1154701,
      "name":"Treadmill, 0% Incline",
      "startTime":"00:25",
      "steps":3783
    }
  ]
}
```

Example: Activity & Exercise Logs: Response

```
"goals":{
  "caloriesOut":2826,
  "distance":8.05,
  "floors":150,
  "steps":10000
},
"summary":{
  "activityCalories":230,
  "caloriesBMR":1913,
  "caloriesOut":2143,
...
```

Who are providing REST interfaces?

- Google Cloud Storage Service
- Google Search REST
- Netflix
- Twitter
- Flickr
- Amazon eCommerce
- Amazon S3
- ...

Part 2. Large scale data storage system

Data Exchange Model

RESTful Service
This material is built based on,


Representational State Transfer (REST)

• An architectural style for networked hypermedia applications
• Used to build Web services that are lightweight, maintainable and scalable

• RESTful service
  • A service based on REST

• REST is not dependent on any protocol
  • But, almost every RESTful service uses HTTP as its underlying protocol
RESTful services

• REST is NOT a standard

• It uses components that are based on standards
  • HTTP
  • URL
  • XML/HTML/GIF/JPEG/etc (Resource Representation)
  • Text/xml, text/html, image/gif, etc (MIME Types)

To be a REST client

• Endpoint

  https://simple-weather.p.mashape.com/aqi
Results (Using Java)

```java
HttpResponse<String> response =
    Unirest.get("https://simple-weather.p.mashape.com/aqi?
    lat=40.57&lng=-105")
    .header("X-Mashape-Key",
        "gaDMJi5MW2mshLzYIAZU8BkLHA6RplzETckjsnzQGZ1IIa9Amw")
    .header("Accept", "text/plain")
    .asString();
```

4 major HTTP methods for REST CRUD

- **Create**, **Read**, **Update**, and **Delete**
  - **POST** – Update
  - **GET** – Read
  - **PUT** – Create
  - **DELETE** – Delete
Part 2. Large scale data storage system

Data Exchange Model

RESTful Service: GET

When to use GET

• Caches depend on the ability to serve cached representations
  • Without contacting the original server
• Safe and idempotent information retrieval

Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of $N > 0$ identical requests is the same as that for a single request.
GET example

# Bookmark a page
Host: www.example.org

# Add an item to a shopping cart
GET /add_cart?pid=1234 HTTP/1.1
Host: www.example.org

# Send a message
GET /message/send?message=I%20am%20reading HTTP/1.1
Host: www.example.org

# Delete a note
GET /notes/delete?id=1234 HTTP/1.1
Host: www.example.org

Designing a Web Service with GET

- If it is not safe to cache
  - Make the response noncacheable
    - Add a Cache-Control: no-cache header

- Consider any possible side effects
- Implement servers which can handle frequently repeatable operations (e.g. concurrent access)
Part 2. Large scale data storage system

Data Exchange Model

RESTful Service: POST

When to use POST

• To create a new resource (sub-resource)
• To run a query with large inputs
• To perform any unsafe or non-idempotent operation (when no other HTTP method is available)
Continued

• Originally, POST was designed for
  • Annotation of existing resources
  • Posting on group articles
    • Creates a child resource
  • Providing append operations for database
  • E.g. Create a resource that lives under /items resource
    • /items/1, /items/2...

• Unsafe and non-idempotent processing for the server

Creating Resources Using POST

• Submit a POST request with a representation of the resource to be created by the factory resource

• Optional Slug header
  • Name of the new resource suggested by clients
**POST request**

```xml
# Request
POST /user/smith HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
Slug: Home Address

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

**POST Response**

```xml
# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Location:
http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self"
       href="http://www.example.org/user/smith/address/home_address"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```
Part 2. Large scale data storage system
Data Exchange Model

RESTful Service: PUT

Creating Resources Using **PUT**

- **PUT** requests that the enclosed *entity be stored under the supplied URI*
- **PUT** is idempotent
- **Use PUT to create/add new resources only when clients can decide URIs of resources**
  - Otherwise, use **POST**
In RFC of HTTP,

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource. If the server desires that the request be applied to a different URI, it MUST send a 301 (Moved Permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

Is PUT idempotent?

- Is DELETE idempotent?
Is **PUT** idempotent? -- Yes

- Is **DELETE** idempotent? -- Yes

What if there are two conflicting **PUTs**?

- HTTP/REST does not require “lock” for these concurrent access.
- REST is **STATELESS**.
**PUT request**

```bash
# Request
PUT /user/smith/address/home_address HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
Slug: Home Address

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

**PUT Response**

```bash
# Response
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Location: http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href=http://www.example.org/user/smith/address/home_address/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```
Part 2. Large scale data storage system

Data Exchange Model

RESTful Service: DELETE

POST example

```xml
# A SOAP message tunneled over HTTP POST
POST /Messages HTTP/1.1
HOST: www.example.org
Content-Type: application/soap+xml; charset=UTF-8

<soap:Envelope xmlns:soap=http://www.w3.org/2001/12/soap-envelope
    soap:encodingStyle = http://www.w3c.org/2001/12/soap-encoding>

<soap:Body xmlns:ns=http://www.example.org/messages>
    <ns:DeleteMessage>
        <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
</soap:Body>

</soap:Envelope>
```
POST example

# A SOAP message tunneled over HTTP POST
POST /Messages HTTP/1.1
HOST: www.example.org
Content-Type: application/SOAP+xml; charset=UTF-8

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    xmlns:encoding="http://www.w3c.org/2001/12/soap-encoding">
    <soap:Body xmlns:ns="http://www.example.org/messages">
        <ns:DeleteMessage>
            <ns:MessageId>1234</ns:MessageId>
        </ns:DeleteMessage>
    </soap:Body>
</soap:Envelope>

Is this a good design?

DELETE

# Using DELETE
DELETE /message/1234 HTTP/1.1
Host: www.example.org
**DELETE response**

- The server creates a new resource and representation indicating the status of the job
  - The client can query [http://www.example.org/task/1](http://www.example.org/task/1) to learn the status of the request

```xml
HTTP/1.1 202 Accepted
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <status> pending </status>
  <atom:link href="http://www.example.org/task/1" rel="self"/>
  <message xml:lang="en"> Your request has been accepted for processing. </message>
  <created> 2009-07-05T03:10:00Z </created>
  <ping-after> 2009-07-05T03:15:00Z </ping-after>
</status>
```

---

**Part 2. Large scale data storage system**

**Data Exchange Model**

RESTful Service: Managing Errors
How to Return Errors

• Error needs to be represented as well
• Errors in the clients’ input
  • 4xx status code
• Error due to server implementation or current state
  • 5xx status code
• Include a Date header
  • The date-time at which the error occurred

Description of Error

• Formatted and localized document (HTML or plain text) included in a body
  • Except for the HEAD method
• Other details can be linked via a Link header or in the body
• Keep the body descriptive
Error Message

# Avoid returning success code with an error in the body.
HTTP/1.1 200 OK
Content-Type: application/xml; charset=UTF-8

<error>
  <message> Account limit exceeded. </message>
</error>

Is this a good Error response?

Error must be handled by software
Include your error code in the Header

- 400 Bad request
- 401 unauthorized
- 403 forbidden
- 404 not found
- 409 conflict
- 410 gone
- 412 precondition failed
- 413 request entity too large
- 415 unsupported media type

Include your error code in the Header

- 500 Internal Server Error
- 503 Service Unavailable
Provide description

- A brief message describing the error condition
- A longer description with information on how to fix, if applicable
- An identifier for the error
- A link to learn more about the error condition, with tips on how to resolve it

Example of a Good Error message

```xml
# Response
HTTP/1.1 409 Conflict
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Date: Wed, 14 Oct 2009 10:16:54 GMT
Link: <http://www.example.org/errors/limits.html>;rel="help"

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Account limit exceeded. We cannot complete the transfer due to insufficient funds in your accounts</message>
  <error-id>321-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>
  <atom:link href="http://example.org/account/1234" rel="http://example.org/rels/transfer/from/">
  <atom:link href="http://example.org/account/5678" rel="http://example.org/rels/transfer/to/">
</error>
```
12/2/2019 and 12/4/2019

CS435 Introduction to Big Data – Fall 2019

THANK YOU