**Today’s topics**

- FAQs
- BigTable: Column-family storage
- Dynamo: Key-Value storage

---

**FAQs**

- TP1 scores are available
  - If your “Problem statement” section is lower than 1, please see the instructor ASAP

---

**Data Storage and flow management**

**Column Family Stores**

- Google Big Table

---

**This material is built based on,**


---

**Topics in BigTable**

1. Data model
2. Locating tablet
3. Data Compaction
4. Data Compression
5. Caching and prefetching
Column Family Stores: Big Table
4. Data Compression

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

Data Differencing vs. Data Compressing
- Using target file and source file
- Data compression
  - Similar but without the use of source file
- Traditionally considered as different from data compression technology
  - Compression can be thought of as a type of differencing in which the source data is empty

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>28931</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>

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=756973
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

Data Differencing

1. Concatenate source and target data.
2. Parse the data from left to right as in LZ'77 but make sure that a parsed segment starts the target data.
3. Start to output when reaching target data

Delta Instruction [1/3]

- Target window (T)
  - Partitioned from target file
  - Non-overlapping sections

- S: source data segment of length s
  - S is completely known when T is being decoded

- S[j] represents the jth byte in S
- T[k] represent the kth byte in T
- Superstring U will be formed by
  - The address of a byte in S or T is referred to by its location in U

Delta Instruction [2/3]

- ADD
  - This instruction has two arguments, a size s and a sequence of s bytes to be copied

- COPY
  - This instruction has two arguments, a size s and an address p in the string U
  - The arguments specify the substring of U that must be copied
  - We shall assert that such a substring must be entirely contained in either S or T

- RUN
  - This instruction has two arguments, a size s and a byte b, that will be repeated s times

Delta Instruction [3/3]

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>
```

COPY 4, 0
ADD 4, w x y z
COPY 4, 4
COPY 12, 24
RUN 4, z

To reconstruct the target window
process one delta instruction at a time and copies the data (either from the source window or the target window being reconstructed)

Results

<table>
<thead>
<tr>
<th>Compression</th>
<th>Bible</th>
<th>Bible+Bible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>4060556</td>
<td>802112</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>3986771</td>
<td>3986782</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, MapReduce, Cassandra, Hadoop, LevelDB, MongoDB, RocksDB, Lucene

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

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

Key Value Stores: Dynamo
This material is built based on,


What Amazon needs (1/2)

- Amazon’s architecture
  - A highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services
  - Storage technologies that are always available

What Amazon needs (2/2)

- Customer should be able to view and add items to the shopping cart even if:
  - The disks are failing
  - Network routes are flapping or,
  - Data centers are being destroyed by tornados

  - Highly available system with failure resilience
    - Small but significant number of servers
    - Network components

  - Failure handling should not impact availability or performance

Overview of Dynamo (1/2)

- Partitions and replicates data using consistent hashing
- Tracks object version to provide consistency
- Uses quorum-like technique to ensure consistency among replicas
- Uses a decentralized synchronization protocol
  - Storage nodes can be added and removed from Dynamo without any manual partitioning or redistribution
- Gossip based distributed failure detection and membership protocol

Overview of Dynamo (2/2)

- Underlying storage technology
  - Amazon’s e-commerce platform

- Handles peak loads,
  - Over 3 million checkouts in a single day
  - Hundreds of thousands of concurrently active sessions
Why Amazon does not store their state in relational database?

- Functionality of RDBMS is not required for Amazon's tasks
  - Store and retrieve data by primary key
  - Does not require complex querying and management

- Excess functionality requires
  - Expensive hardware
  - Highly skilled personnel

- Hard to achieve scale-out databases

System Assumptions (1/2)

- Query model
  - Simple read and write operations to a data item
    - Uniquely identified by a key
  - State is stored as binary objects (i.e. blobs) identified by unique keys
    - Usually less than 1MB
    - No operations span multiple data items
    - No need for relational schema

- ACID Properties
  - Dynamo targets applications that operate with weaker consistency if this results in high availability
  - Dynamo does not provide any isolation guarantees

System Assumptions (2/2)

- Efficiency
  - The system needs to function on a commodity hardware infrastructure
  - Stringent latency requirements

Summary of techniques

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Partitioning</td>
<td>Consistent Hashing</td>
</tr>
<tr>
<td>(2) High availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
</tr>
<tr>
<td>(3) Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
</tr>
<tr>
<td>(4) Recovering from permanent failures</td>
<td>Anti-entropy using Merkle tree</td>
</tr>
<tr>
<td>(5) Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection</td>
</tr>
</tbody>
</table>

Partitioning algorithm

- Dynamically partitions the data over the set of nodes
- Distributes the load across multiple storage hosts
Key Value Stores: Dynamo

(1) Partitioning
Hash Functions and Their Uses

Hash function

- What is a hashtable?
- Hash function
  - Takes a hash-key value as an argument
  - Produces a bucket number as a result
- Bucket number
  - \(0 \leq (B-1)\), where \(B\) is the number of buckets
- Selecting prime number as the bucket number will reduce the probability of collisions

Hash function and indexing (1/2)

- Index
  - Data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects
  - Index can be one of the fields
    - (name, address, phone) triples and index on the phone field
    - Given phone number, the index allows you to find the record quickly

Hash function and indexing (2/2)

- Hash function is useful to implement indexing
  - A hash function is applied to the value of the hash-key
  - One of the fields
  - Record is placed in the bucket whose number is determined by the hash function
  - The bucket could be a list of records in main-memory, or a disk block

What if hash-keys are not integers?

- String
  - Convert common types of integers using ASCII or Unicode equivalent
- Multi-dimension?
  - e.g. a geospatial coordinate?

Case study: Geohash

- Latitude/longitude geocode system
  - Provides arbitrary precision
    - By selecting the length of the code gradually
  - Colorado State University
    - 40.5748° N, 105.0810° W
    - Geohash: 9xjqbdqm5h3y1
Encoding (1/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Phase 1. Create interleaved bit string `geobits[]`
  - Even bits are from longitude code, LON
  - Odd bits are from latitude code: LAT
- Step 1.
  - If \(-180 \leq \text{LON} \leq 0\) set `geobits[0]` as 0
  - If \(0 < \text{LON} < 180\) set `geobits[0]` as 1

geobits[0] = 0
geobits[1] = 1

Encoding (2/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Step 3.
  - Since `geobits[0]` = 0,
    - If \(-180 \leq \text{LON} < -90\) set `geobits[2]` as 0
    - If \(-90 \leq \text{LON} < 0\) set `geobits[2]` as 1

geobits[2] = 0

Encoding (3/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Step 3.
  - Since `geobits[2]` = 0,
    - If \(-180 \leq \text{LON} < -135\) set `geobits[4]` as 0
    - If \(-135 \leq \text{LON} < -90\) set `geobits[4]` as 1

geobits[4] = 1

Encoding (4/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Step 3.
  - Since `geobits[3]` = 0
    - If \(0 \leq \text{LAT} < 22.5\) set `geobits[5]` as 0
    - If \(22.5 \leq \text{LAT} < 45\) set `geobits[5]` as 1

geobits[5] = 1
Encoding (4/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Repeat this process until your precision requirements are met
- Currently geohash bit string is 010011
- Phase 2.
  - Encode every five bits from the left side, by mapping to the table.
  - First five bits "00000" are mapped to "0"

<table>
<thead>
<tr>
<th>Decimal</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 32</td>
<td>0-9</td>
<td>a-b</td>
<td>c-d</td>
<td>e-f</td>
<td>g-</td>
<td>h-i</td>
<td>j-k</td>
<td>l-m</td>
<td>n-o</td>
<td>p-q</td>
</tr>
</tbody>
</table>

Key Value Stores: Dynamo

(1) Partitioning

Consistent Hashing

Partitioning scheme

- Dynamo is a zero-hop DHT
  - Each node maintains enough routing information locally
- Inspired by Consistent Hashing and Chord

Non-consistent hashing vs. consistent hashing

- When a hash table is resized
  - Non-consistent hashing algorithm requires to re-hash complete table
  - Consistent hashing algorithm requires only partial records of the table

Key 3 will be stored in machine successor(3) = 5

Key 2 will be stored in machine successor(2) = 5

Key 1 will be stored in machine successor(1) = 1
Consistent hashing (3/3)

If machine C leaves circle, Successor(5) will point to A
If machine N joins circle, successor(2) will point to N
New node N

Scalable Key location

- In consistent hashing:
  - Each node need only be aware of its successor node on the circle
  - Queries can be passed around the circle via these successor pointers until it finds resource
- What is the disadvantage of this scheme?
  - It may require traversing all N nodes to find the appropriate mapping