PART 2.
DATA STORAGE AND FLOW MANAGEMENT

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics
• FAQs
• BigTable
• Dynamo

FAQs
• Help session for PA3
  • April 8th 11:00AM – noon CSB130
  • Please check the assignment page for the link

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

- 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

Example of the Constitution of the US and the King James Bible

<table>
<thead>
<tr>
<th>File</th>
<th>Text</th>
<th>gap</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>99446</td>
<td>26631</td>
<td>1.911</td>
</tr>
<tr>
<td>Bible</td>
<td>446056</td>
<td>121495</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
  - $<\text{start}, \text{length}>$
  - 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, ..."

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>Compression</th>
<th>Bible</th>
<th>Bible+Bible</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>1321495</td>
<td>2642389</td>
</tr>
<tr>
<td>com50</td>
<td>3660771</td>
<td>3906782</td>
</tr>
<tr>
<td>com20</td>
<td>1318699</td>
<td>1362422</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

Column Family Stores: Big Table
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
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
- 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

If you design a data storage system for,

Amazon.com to store transaction data for the shopping cart management, how would you prioritize properties: Consistency, Availability, or Partition tolerance? And Why?

What Amazon needs

- 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>Sticky 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>

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
Partitioning algorithm
- Dynamically partitions the data over the set of nodes
- Distributes the load across multiple storage hosts
- Dynamo uses consistent hashing
  - Relies on the Chord Distributed Hash Table

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
  - $0 \sim (B-1)$, 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: 9vjqbdqm5h3y1

Resolutions with length of geohash string

Resolutions with length of geohash string

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<=LON<=0 set geobits[0] as 0
  - If 0<LON<=180 set geobits[0] as 1

Encoding (2/4)

- LAT: 40.5747652 LON:-105.0865006 (CSU)
- Step 3.
  - Since geobits[0] =0,
  - If -180<=LON<=-90 set geobits[2] as 0
  - If -90<LON<=0 set geobits[2] as 1
- Step 4.
  - Since geobits[1] = 1
  - If 0<=LAT<=45 set geobits[3] as 0
  - If 45< LAT<90 set geobits[3] as 1
Encoding (3/4)

- LAT: 40.5747652 LON: -105.0865006 (CSU)
- Step 3.
  - Since geobits[2] = 0,
  - If -180 <= LON <= -135 set geobits[4] as 0
  - If -135 < LON <= -90 set geobits[4] as 1
- Step 4.
  - Since geobits[3] = 0
  - If 0 <= LAT <= 22.5 set geobits[5] as 0
  - If 22.5 < LAT < 45 set geobits[5] as 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 01001

Consistent Hashing

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
Consistent hashing (3/3)

Inconsistent 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

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

Scalable Key location in Chord

- Let $m$ be the number of bits in the key/node identifiers
- Each node $n$, maintains,
  - A routing table with (at most) $m$ entries
  - Called the finger table

- The $i$th entry in the table at node $n$, contains the identity of the first node, $x$.
  - Succeeds $n$ by at least $2^i$ on the identifier circle
  - i.e. $x = \text{successor } (n+2^i)$, where $i \leq m$ (and all arithmetic is modulo $2^m$)

Definition of variables for node $n$, using $m$-bit identifiers

- $\text{Finger}[i].\text{start} = (n+2^i) \mod 2^m$, $1 \leq i \leq m$
- $\text{Finger}[i].\text{interval} = [\text{finger}[i].\text{start}, \text{finger}[i+1].\text{start})$
- $\text{Finger}[i].\text{node} = \text{first node} \geq n.\text{finger}[i].\text{start}$
- Finger table
  - The Chord identifier
  - The IP address of the relevant node
  - First finger of \(n\) is its immediate successor on the circle

### Lookup process (1/3)

- Each node stores information about only a small number of other nodes
- A node’s finger table generally does not contain enough information to determine the successor of an arbitrary key \(k\)
- What happens when a node \(n\) does not know the successor of a key \(k\)?
  - If \(n\) finds a node whose ID is closer to its own than \(k\), that node will know more about the identifier circle in the region of \(k\) than \(n\) does

### Lookup process (2/3)

- \(n\) searches its finger table for the node \(j\)
  - Whose ID most immediately precedes \(k\)
  - Ask \(j\) for the node it knows whose ID is closest to \(k\)
  1. Go clockwise
  2. Never overshoot

### Lookup process (3/3)

0. Request comes into node 3 to find the successor of ID 1.
1. Node 3 wants to find the successor of identifier 1
2. Identifier 1 belongs to [7, 3)
3. Check succ: 3
4. Node 3 asks node 0 to find successor of 1
5. Successor of 1 is 1
6. Request comes into node 3 to find the successor of identifier 1
7. Node 3 wants to find the successor of identifier 1
8. First finger of 3 is its immediate successor on the circle

### Lookup process: example 1

0. Request comes into node 1 to find the successor of ID 4.
1. Node 3 wants to find the successor of identifier 1
2. Identifier 4 belongs to [4, 5)
3. Check succ: 3
4. Node 1 asks node 3 to find successor of 4
5. Successor of 4 is 0
6. Request comes into node 1 to find the successor of ID 4.
7. Node 3 wants to find the successor of identifier 1
8. First finger of 3 is its immediate successor on the circle
Lookup process: example 2

0. Request comes into node 3.
1. Node 3 wants to find the successor of identifier 0
2. Identifier 0 belongs to [7,3)
3. Check succ: 0
4. Node 3 asks node 0 to find successor of 1
5. Machine is using identifier 0 as well, succ is 0.

Dynamo’s partitioning

- Inspired by Consistent Hashing and Chord
- When a node starts for the first time
  - Chooses its set of tokens (virtual nodes in the consistent hash space)
  - Maps nodes to their respective token sets
  - Stores both tokens and nodes onto disk
- Repeated reconciliation of the membership change
- Partitioning and placement information are propagated via the gossip-based protocol
  - Token ranges handled by its peers
- Direct forwarding of read/write operations are possible