Today’s topics

• No SQL storage

This material is built based on,


FAQs

• Term project presentations
  • 4/30 (6 teams), 5/2 (6 teams), 5/4 (4 teams)
  • Please attend at least 2 presentation sessions and ask questions or provide comments
  • Participation score (attendance + question)

• 4/25 Wednesday
  • Data exchange

• Quizzes

• Final exam overview

Part 2. Large scale data storage system

NoSQL Storage: 2. Column Family Stores

Google’s Big Table
System Structure

Building blocks

- Memtable: in-memory table
  - utf-8 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 (1/2)

- 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

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 kept in a 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 tables
  - **3rd level**: metadata table holds locations of tablets of all other tables. Metadata table itself can be split into multiple tablets.

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 [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 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 server locations [4/4]

- **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 stale
    - 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 stale
    - 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` 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

**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

Existing approaches to improve performance
• Hash
• B+ tree
• External file: create separate hash or tree index

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

• 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 footprint
  • Improved write performance
  • Read performance is still slightly poorer than B+ tree
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

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

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 support for values 1. It is dictionary for value 0-1
  - 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

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>8920112</td>
<td>2642389</td>
<td>1.9995</td>
</tr>
<tr>
<td>Bible+bible</td>
<td>1321495</td>
<td>8551124</td>
<td>1.0</td>
</tr>
</tbody>
</table>


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, ...”

Implementation of Data compression

- 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>Test 1</th>
<th>Test 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const+Const</td>
<td>0.5658</td>
<td>2631</td>
</tr>
<tr>
<td>Bible+file</td>
<td>0.571</td>
<td>263389</td>
</tr>
<tr>
<td>Bible</td>
<td>0.567</td>
<td>263212</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compression</th>
<th>Linux</th>
<th>Bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>snappy</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Snappy

- Based on L277
  - 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

### 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