FAQs
• Term project presentations
  • 11/28 (5 teams), 12/3 (5 teams), 5/5 (5 teams)
  • Please attend presentation sessions and ask questions or provide comments
    • Participation score (attendance(3) + question(2))
  • 10 minutes (5-7 slides)
    • Title
    • Introduction (motivation, background)
    • Methodology (1-2)
    • Evaluation and analysis (1-2)
    • Conclusion
  • 2 minutes (Q&A)
    • All team members MUST participate in the presentation
    • Otherwise, there will be a score reduction

Today’s topics
• No SQL storage

Part 2. Large scale data storage system
NoSQL Storage: 2. 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

Scalability and latency

- Scale in capacity
  - E.g., webtable
    - 100,000,000,000 pages * 10 versions per page * 20KB/verison = 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

- 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

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

Example of data model with Webtable

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

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 100M 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
- 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

- 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 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

- Each cell in BigTable can contain multiple versions of the same data
  - Indexed by timestamp
- BigTable timestamp
  - 64-bit integer
  - Assigned by BigTable
  - Real-time in microsecond
  - 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
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

Google’s Big Table

Data model

Locating tablet

Data Compaction

Data Compression

Caching and prefetching

Building blocks

- Memtable: 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

SSTable: Sorted String Table

Reading and writing data can dominate running time

Random reads and writes are critical features

- 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/
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 a pointer to relevant server
  • 1st level: bootstrapped from Chubby (lock service), points to 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

Caching the tablet locations [1/5]

• 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 [2/5]

• 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

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)?

• 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 round trips 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 round trips
      - To the metadata tablet server (it misses tablet info due to the stale info) (c-1)
      - To the root server to retrieve the location of the metadata tablet (b-1-2)
      - To the metadata tablet server to retrieve tablet server location (c-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
      - Chubby creates a uniquely-named file in a specific Chubby directory
    - Exclusive lock
    - Master monitors this directory to discover tablet servers
  - 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
  - Release its lock
  - Master will reallocate its tablets more quickly
Tablet status

- The persistent state of a tablet is stored in GFS

Tablet Representation

Tablet status

- The persistent state of a tablet is stored in GFS

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
  - 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

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)