Bigtable - A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

Presented by

Ripul Jain
Bigtable

- A distributed storage system for managing structured data
- Developed by Google to deal with massive amount of data
- Not a relational database
- Designed for
  - Wide Applicability
  - Scalability
  - High Performance
  - High Availability
Data Model

- A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map.

- Map is indexed by a row key, column key, and a timestamp:
  - \((\text{row:}\text{string}, \text{column:}\text{string}, \text{time:}\text{int64}) \rightarrow \text{string}\)

**Figure 1:** A slice of example table that stores webpages

Source: Chang et al., OSDI 2006
Rows and Columns

➤ Row key is an arbitrary string
  - Access to data under a single row key is atomic
  - Rows maintained in sorted lexicographic order

➤ Columns grouped into column families
  - Column key = family:qualifier
  - Data within a column family is of same type
  - Unbounded number of columns in a table

➤ Timestamps are used to store different versions of same data in a cell
  - Clients can specify the last n versions they want to keep
Tables

- Rows ranges in the table are dynamically partitioned into Tablets
  - Each Tablet is about 100 - 200 Mb in size
  - Unit of distribution and load balancing

- Each machine is responsible for 10 to 1000 tablets
  - Not necessarily contiguous range of rows/tablets
  - Master can migrate tablets away from overloaded or failed machines
  - Fast Recovery: 100 machines can pick up 1 tablet from a failed 100 tablet machine
API

- Bigtable API provides functions for creating and deleting table and column families.

- Supports single-row transactions, which can be used to perform atomic read-modify-write sequences under a single row key.

- Can retrieve data from a single row or all rows.

- Can retrieve data for all columns, certain column families or specific columns.

- Supports execution of client-supplied scripts on the tablet servers.
Bigtable Building Blocks

- **Google File System (GFS)**
  - Used to store log and data files

- **SSTable**
  - Persistent ordered immutable map from keys to values
  - Sequence of 64KB blocks plus an index for block lookup
  - Can be mapped into main memory
  - Multiple SSTables form a Tablet

- **Chubby**
  - A highly available, persistent and distributed lock service
  - Stores the location of Bigtable data
  - Store Bigtable schema
  - Ensures there’s only 1 Bigtable master
  - Keeps track of tablet servers
System Structure

- **Bigtable Master**
  - Assigns tablets to tablet servers
  - Detects addition and expiration of tablet servers
  - Performs load balancing
  - Handles garbage collection
  - Handles schema changes such as table and column family creations

- **Bigtable Tablet Servers**
  - Manages a set of tablets
  - Handles read and write requests to loaded tablets
  - Splits tablets that have grown too large
Locating Tablets

- Three-level hierarchical lookup scheme for tablets

Figure 2: Tablet location hierarchy
Tablet Assignment

- Each tablet is assigned to one tablet server at a time
  - Tablet server maintains an exclusive lock on a newly created file in a specific Chubby directory

- Master scans the servers directory to monitor tablet servers
  - Communicates with tablets servers to find tablet assignments
  - If root tablet not found, adds root tablet to unassigned tablets
  - Periodically asks each tablet server for the status of its lock
  - Keeps track of unassigned tablets and handles their assignment

- Changes to tablet structure
  - Master initiates Table creation/deletion
  - Master initiates tablet merge
  - Tablet server initiates tablet splits, updates METADATA table and notifies master
Figure 3: Tablet Representation

Source: Chang et al., OSDI 2006
Tablet Serving

- **Write operation**

  Tablet server ensures the client has sufficient privileges for write operation. Writes are placed in the commit log first on GFS. After the write update has been committed, its contents are mirrored into an in-memory sorted buffer called memtable. After a memtable grows to a certain size, it gets written to GFS.

- **Read operation**

  Tablet server ensures the client has sufficient privileges for read operation. Reads then are executed on a merged view of sequence of SSTables and memtable. The paper doesn’t mention how often this is done.
Compactions

- **Minor compaction**
  - When a memtable grows too big, it is frozen, converted to SSTable and written to GFS
  - Reduces memory usage of the tablet server
  - Reduces the amount of data that needs to be read from commit log during recovery

- **Merging compaction**
  - Reads content of few SSTables and the memtable, and writes out a new SSTable
  - Applied periodically

- **Major compaction**
  - A merging compaction that results in exactly one SSTable
  - No deletion records
  - Applied regularly to all tablets
Refinements

- Locality groups
  - Column families can be assigned to a locality group
  - Separate SSTable for each locality group
  - Used to better organize data for performance
  - Can be declared to be in-memory

- Compression
  - SSTables for locality groups can be compressed
  - Clients can specify which compression format to use

- Caching for read performance
  - Scan cache caches the key-value pairs returned by SSTable interface to tablet server
  - Block cache caches SSTable blocks that are read from GFS
Refinements

- Bloom filters for SSTables
  - A read operation may need to read many SSTables
  - Does (row, column) exist in the tablet?
  - Reduces disk seeks for read operations

- Exploiting immutability
  - SSTables are immutable
  - Master removes the obsolete SSTables as a mark-and-sweep garbage collection
  - Memtable is the only mutable data structure
  - Use copy-on-write semantics to allow parallel reads and writes

- Shared commit log
  - Large number of log files written concurrently in GFS leads to poor performance
  - Idea: Single physical log file per tablet server
  - Complicates shared log recovery
Shared Log Recovery

- Idea: Partition and sort the commit log entries in order of keys <table, row, log sequence number>

- Tablet server notifies master of log chunks they need to read

- Tablet Servers do the sorting and master coordinates which sorted chunks are needed by which tablet servers

- Tablets server read sorted data for its tablets directly from its peers
Performance Evaluation

Setup
- 1786 machines with two 400 GB hard drives each, large physical memory
- Each tablet server has 1 GB memory
- No. of clients = No. of servers

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>random reads</td>
<td>1212</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
</tr>
</tbody>
</table>

Figure 4: No. of 1000-byte values read/written per second per tablet server
Performance Evaluation

Figure 5: Aggregate rate
### Figure 6: Characteristics of a few tables in use at Google

<table>
<thead>
<tr>
<th>Project name</th>
<th>Size (TB)</th>
<th>Comp. ratio</th>
<th># (B) Cells</th>
<th># Families</th>
<th># Groups</th>
<th>% MMap</th>
<th>Frontend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl</td>
<td>800</td>
<td>11%</td>
<td>1000</td>
<td>16</td>
<td>8</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Crawl</td>
<td>50</td>
<td>33%</td>
<td>200</td>
<td>2</td>
<td>2</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Analytics</td>
<td>20</td>
<td>29%</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0%</td>
<td>Yes</td>
</tr>
<tr>
<td>Analytics</td>
<td>200</td>
<td>14%</td>
<td>80</td>
<td>1</td>
<td>1</td>
<td>0%</td>
<td>Yes</td>
</tr>
<tr>
<td>Base</td>
<td>2</td>
<td>31%</td>
<td>10</td>
<td>29</td>
<td>3</td>
<td>15%</td>
<td>Yes</td>
</tr>
<tr>
<td>Earth</td>
<td>0.5</td>
<td>64%</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>33%</td>
<td>Yes</td>
</tr>
<tr>
<td>Earth</td>
<td>70</td>
<td>–</td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>0%</td>
<td>No</td>
</tr>
<tr>
<td>Orkut</td>
<td>9</td>
<td>–</td>
<td>0.9</td>
<td>8</td>
<td>5</td>
<td>1%</td>
<td>Yes</td>
</tr>
<tr>
<td>Pers. Search</td>
<td>4</td>
<td>47%</td>
<td>6</td>
<td>93</td>
<td>11</td>
<td>5%</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Size (measured before compression) and # Cells indicate approximate sizes. Comp. ratio (compression ratio) is not given for tables that have compression disabled. Frontend indicates that the application’s performance is latency-sensitive.*
Lessons

- Large distributed systems are vulnerable to many types of failures
- Large systems need proper system-level monitoring
- Value simple designs
- Delay adding new features until it is clear how the new features are going to be used
Critique

▶ Strengths

- Flexible data model
- Able to handle mixed read-write workloads
- Demonstrated that single-master systems can avoid bottlenecks and scale well

▶ Weaknesses

- No support for multi-row transactions (added in 2008)
- Stability of the system is dependent on Chubby
- No performance comparison with other databases
- Bigtable is difficult for new users to adapt to, specially if they are accustomed to relational databases
Competition

- Public version of Bigtable called Google Cloud Bigtable released in 2015
- Apache HBase is modelled after Bigtable
- IBM DB2 Parallel Edition
- Sybase IQ
- C-Store
- Apache Cassandra
- LevelDB
Conclusion

- Excellent example of how to design a storage system for large scale data that needs to be frequently read or searched.

- Self managing, as servers can be dynamically added or removed

- Can be used as Mapreduce input or output

- Tried and tested extensively in production

- Can be extended to Bigtable as a service

- Leaves room for future work like support for secondary indices