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

Google, Inc.
Introduction
Data Model
API
Building Blocks
  • SSTable
Implementation
Tablet Location
Tablet Assignment
Tablet Serving
Compactions
Refinements
Performance Evaluation
Lessons
Related Works
Conclusions
Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.
Data Model

Rows

- The row keys in a table are arbitrary strings: Currently up to 64KB in size
- Every read or write of data under a single row key is atomic
- Each row range is called a tablet, which is the unit of distribution and load balancing.
**Data Model**

- **Column Families**

  - Column keys are grouped into sets called *column families*, which form the basic unit of access control.
  
  - A column key is named using the following syntax: `family:qualier`.
  
  - *Column family names must be printable*, but qualifiers may be arbitrary strings.
Data Model

- **Timestamps**

  - Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp.
Data Model
API

- Creating and deleting tables and column families.
- Changing cluster, table, and column family metadata, such as access control rights.
- Single-row transactions, no general transactions
- Client-supplied scripts, limited usage.
- Scripts written in Sawzall
Building Blocks

- Bigtable uses the distributed **Google File System (GFS)** to store log and data files.

- The Google **SSTable** file format is used internally to store Bigtable data.
SSTable

- Key/value pairs in a specified key range
- SSTable contains a sequence of blocks.
- A block index is used to locate blocks
- Binary search in index
- SSTable can be completely mapped into memory
Bigtable relies on a highly-available and persistent distributed lock service called Chubby

Chubby has five active replicas, one of them is master.

Chubby uses the Paxos algorithm to keep its replicas consistent in the face of failure.
The Bigtable implementation has three major components:

- A library that is linked into every client
- One master server
- Many tablet servers
Tablet Location

A three-level hierarchy analogous to that of a B+ tree to store tablet location information
The first level is a file stored in Chubby that contains the location of the root tablet.

The root tablet contains the location of all tablets in a special METADATA table.

Each METADATA tablet contains the location of a set of user tablets.

The root tablet is just the first tablet in the METADATA table, but is treated specially.

The client library caches tablet locations.
Tablet Assignment

- Bigtable uses Chubby to keep track of tablet servers via master.
- The master assigns the tablet by sending a tablet load request to the tablet server.
- If a tablet server loses its exclusive lock, it can attempt to reacquire an exclusive lock.
- If the file no longer exists, the tablet server kills itself.
- When a tablet server terminates, it attempts to release its lock so that master reassigns its tablets more quickly.
When a master is started by the cluster management system,
The master grabs a unique *master lock in Chubby, which prevents concurrent* master instantiations.
The master scans the servers directory in Chubby to find the live servers.
The master communicates with every live tablet server to discover what tablets are already assigned to each server.
The master scans the METADATA table to learn the set of tablets.
Tablet Serving

Memory

GFS

tablet log

Write Op

memtable

Read Op

SSTable Files
Compactions

- As write operations execute, the size of the memtable increases.

- When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS.

- This is called *minor compaction*. 
A *merging compaction* reads the contents of a few SSTables and the memtable, and writes out a new SSTable.

The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. 
Clients can group multiple column families together into a *locality group*.

More efficient reads

Page metadata (such as language and checksums) can be in one locality group, an application that wants to read the metadata does not need to read through all of the page contents.
Many clients use a two-pass custom compression scheme.

The first pass uses Bentley and McIlroy's scheme.

The second pass uses a fast compression algorithm.

This two-pass compression scheme achieved a 10-to-1 reduction in space.
Refinements: Commit-log implementation

To avoid duplicating log reads by first sorting the commit log entries in order of the keys: (table; row name; log sequence number).

As a result, logs can be read efficiently with one disk seek followed by a sequential read.

Also, to protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file.
Performance Evaluation

- Read the string stored under the row key
- Write is same as read but it writes string under the row key
- The *random read benchmark shadowed the operation of* the random write benchmark
The *scan benchmark* is similar to the *sequential read* benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range.

The *random reads (mem) benchmark* is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, so no read from GPS.
Performance Evaluation

The table shows the rate per tablet server

<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>
Performance Evaluation

[Graph showing performance evaluation with number of tablet servers on the x-axis and values read/written per second on the y-axis. Different lines represent scans, random reads (mem), random writes, sequential reads, sequential writes, and random reads.]

Values read/written per second

Number of tablet servers
Performance Evaluation

- Random reads are slower than all other operations by an order of magnitude or more.
- Random and sequential writes perform better than random reads since writes to a single commit log and uses group commit to stream.
- Sequential reads perform better than random reads.
- Scans are even faster since the tablet server can return a large number of values in response.
### Real Applications

<table>
<thead>
<tr>
<th>Project name</th>
<th>Table size (TB)</th>
<th>Compression ratio</th>
<th># Cells (billions)</th>
<th># Column Families</th>
<th># Locality Groups</th>
<th>% in memory</th>
<th>Latency-sensitive?</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>Google 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>Google 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>Google 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>Google 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>Google 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>Personalized 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>

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.
Lessons

- Large distributed systems are vulnerable to many types of failures: Memory and network corruption, extended and asymmetric network partitions, overflow of GFS quotas.

- Simple design: Whatever the code or the system is huge or not, simple design help in code maintenance and debugging.
Related Work

- The Boxwood project
- Oracle's Real Application Cluster database and IBM's DB2 Parallel Edition provide a complete relational model with transactions
- C–Store and Bigtable share many characteristics but C–Store behaves like relational database
Conclusions

- Bigtable, a distributed system for storing structured data at Google

- As of August 2006, more than sixty projects are using Bigtable.

- Different from usual relational database with its design and interface