Bigtable: A Distributed Storage System for Structured Data
By Fay Chang, et al. OSDI 2006

Presenter: Xiang Gao
Mar. 28 2013
Outline

- Motivation
- Data Model
- APIs
- Building Blocks
- Implementation
- Refinement
- Evaluation
Google’s Motivation

- Lots of data
  - Web contents, satellite data, user data, email, etc.
  - Different projects/applications
  - Hundreds of millions of users
  - Many incoming requests

- Storage for structured data

- No commercial system big enough

- Low-level storage optimization help performance significantly
Bigtable

- Distributed multi-level map
- Fault-tolerant, persistent
- **Scalable**
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- **Self-managing**
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance
Data Model

- A sparse, distributed persistent multi-dimensional sorted map

- The map is indexed by a row key, a column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row, column, timestamp) \(\rightarrow\) cell contents
Data Model

- **Rows**
  - Arbitrary string
  - Access to data in a row is atomic
  - Ordered lexicographically
Data Model

- Column
  - Two-level name structure:
    - family: qualifier
  - Column Family is the unit of access control
Data Model

- Timestamps
  - Store different versions of data in a cell
  - Lookup options
    - Return most recent K values
    - Return all values

![Diagram showing timestamps and data model concepts]
Data Model

- The row range for a table is dynamically partitioned
- Each row range is called a **tablet**
- Tablet is the unit for distribution and load balancing
APIs

- Metadata operations
  - Create/delete tables, column families, change metadata

- Writes
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row

- Reads
  - Scanner: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns
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
- An SSTable provides a persistent, ordered immutable map from keys to values
  - Each SSTable contains a sequence of blocks
  - A block index (stored at the end of SSTable) is used to locate blocks
  - The index is loaded into memory when the SSTable is open
Building Blocks

Cluster scheduling master
Chubby Lock service
GFS master

Machine 1
User app1
User app2
BigTable server
Scheduler slave
GFS chunkserver
Linux

Machine 2
User app1
BigTable server
Scheduler slave
GFS chunkserver
Linux

Machine N
BigTable master
Scheduler slave
GFS chunkserver
Linux
Tablet and SSTables

- Contains some range of rows of the table
- Built out of multiple **SSTables**

<table>
<thead>
<tr>
<th>Tablet</th>
<th>**Start:**alpha</th>
<th>**End:**apple</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K block</td>
<td>64K block</td>
<td>64K block</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>SSTable</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Index</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64K block</th>
<th>64K block</th>
<th>64K block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>SSTable</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Index</strong></td>
</tr>
</tbody>
</table>
Chubby

- \{lock/file/name\} service
- Coarse-grained locks
- Each clients has a session with Chubby.
  - The session expires if it is unable to renew its session lease within the lease expiration time.
- 5 replicas, need a majority vote to be active
  - Service is functional when majority of the replicas are running and in communication with one another – when there is a quorum

- Also an OSDI ’06 Paper
Implementation

- Single-master distributed system
- Three major components
  - Library that linked into every client
  - One master server
    - Assigning tablets to tablet servers
    - Detecting addition and expiration of tablet servers
    - Balancing tablet-server load
    - Garbage collection
    - Metadata Operations
  - Many tablet servers
    - Tablet servers handle read and write requests to its table
    - Splits tablets that have grown too large
Implementation

BigTable

BigTable Master
Performs metadata ops and load balancing

BigTable Tablet Server
Serves data

Cluster scheduling system
Handles failover, monitoring

BigTable Tablet Server
Serves data

GFS
Holds tablet data, logs

Chubby
Holds metadata, handles master election

BigTable Client

BigTable Client Library
Each Tablets is assigned to one tablet server.
  ◦ Tablet holds contiguous range of rows
    • Clients can often choose row keys to achieve locality
  ◦ Aim for 100MB to 200MB of data per tablet

Tablet server is responsible for 100 tablets
  ◦ Fast recovery:
    • 100 machines each pick up 1 tablet for failed machine
  ◦ Fine-grained load balancing:
    • Migrate tablets away from overloaded machine
    • Master makes load-balancing decisions
Given a row, how do clients find the location of the tablet whose row range covers the target row?

- METADATA: Key: table id + end row, Data: location
- Aggressive Caching and Prefetching at Client side
A 3-level hierarchy analogous to that of a B+ tree to store tablet location information:

- A file stored in chubby contains location of the root tablet
- Root tablet contains location of Metadata tablets
  - The root tablet never splits
- Each meta-data tablet contains the locations of a set of user tablets

- Client reads the **Chubby file** that points to the root tablet
  - This starts the location process

- Client library caches tablet locations
  - Moves up the hierarchy if location N/A
When a tablet server starts, it creates and acquires exclusive lock on, a uniquely-named file in a specific Chubby directory
  ◦ Call this servers directory

A tablet server stops serving its tablets if it loses its exclusive lock
  ◦ This may happen if there is a network connection failure that causes the tablet server to lose its Chubby session
Tablet Server

- A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists.
- If the file no longer exists then the tablet server will never be able to serve again:
  - Kills itself
  - At some point it can restart; it goes to a pool of unassigned tablet servers.
Master Operation

- Upon start up the master needs to discover the current tablet assignment.
  - Obtains unique master lock in Chubby
    - Prevents concurrent master instantiations
  - Scans *servers directory* in Chubby for live servers
  - Communicates with every live tablet server
    - Discover all tablets
  - Scans METADATA table to learn the set of tablets
    - Unassigned tablets are marked for assignment
Master Operation

- Detect tablet server failures/resumption
- Master periodically asks each tablet server for the status of its lock
- Tablet server lost its lock or master cannot contact tablet server:
  - Master attempts to acquire exclusive lock on the server’s file in the servers directory
  - If master acquires the lock then the tablets assigned to the tablet server are assigned to others
- If master loses its Chubby session then it kills itself
  - Election will be triggered
Tablet Server Failover

Logical view:

Master -> Chubby Server

Physical layout:
Tablet server
GFS Chunkserver
SSTable SSTable SSTable
Tablet Tablet Tablet

Message sent to tablet server by master
(Other tablet servers drafted to serve other “abandoned” tablets)
Backup copy of tablet made primary
Extra replica of tablet created automatically by GFS
- Commit log stores the updates that are made to the data
- Recent updates are stored in memtable
- Older updates are stored in SSTable files
Tablet Server

- Recovery process
  - Metadata contains SSTables and redo points
- Reads/Writes that arrive at tablet server
  - Well-formedness
  - Authorization: Chubby holds the permission list
  - Group commit
Compactions

- **Minor compaction** – convert the memtable into an SSTable
  - At the threshold
  - Reduce memory usage
  - Reduce log traffic on restart
- **Merging compaction**
  - Periodically
  - Reduce number of SSTables
  - Good place to apply policy “keep only N versions”
- **Major compaction**
  - Results in only one SSTable
  - No deletion records, only live data
Refinements

- **Locality groups**
  - Clients can group multiple column families together into a *locality group*.

- **Compression**
  - Compression applied to each SSTable block separately
  - Uses *Bentley and McIlroy's* scheme and *fast compression* algorithm

- **Caching for read performance**
  - Scan Cache and Block Cache

- **Bloom filters**
  - Reduce the number of disk accesses
Refinements

- **Commit-log implementation**
  - One log per tablet server rather than one log per tablet

- **Speeding up tablet recovery**
  - Minor compaction when tablet moves

- **Exploiting SSTable immutability**
  - No need to synchronize accesses to file system when reading SSTables
  - Efficient concurrency control -- over rows
  - Deletes work like garbage collection on removing obsolete SSTables
  - Enables quick tablet split: parent SSTables used by children
## Evaluation

<table>
<thead>
<tr>
<th>Experiment</th>
<th>1</th>
<th>50</th>
<th>250</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>random reads</td>
<td>1212</td>
<td>593</td>
<td>479</td>
<td>241</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
<td>8511</td>
<td>8000</td>
<td>6250</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
<td>3745</td>
<td>3425</td>
<td>2000</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
<td>2463</td>
<td>2625</td>
<td>2469</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
<td>3623</td>
<td>2451</td>
<td>1905</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
<td>10526</td>
<td>9524</td>
<td>7843</td>
</tr>
</tbody>
</table>
As the number of tablet servers is increased by a factor of 500:

- Performance of random reads from memory increases by a factor of 300.
- Performance of scans increases by a factor of 260.

Not Linear!
WHY?
Critiques

- No detailed argument about how the imbalance in load prevents good scaling.
- The authors claim a very low failure rate, whereas they also mentioned the vulnerability in lessons due to many types of failures, I would like to see how they improve the failure rate and corresponding data.
- The API does not support standard SQL query, which may complicate the application.
Thanks