BigTable

Tom Anderson
(slides from Jeff Dean and Dan Ports)

Outline

Last time:
  – Chubby: Paxos based lock server, service coordination, dynamic configuration manager

Today/Monday:
  – BigTable: scalable storage of structured data
  – GFS: large-scale storage for bulk data
BigTable Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    • Contents, crawl metadata, links, anchors, pagerank, ...
  - Per-user data:
    • User preference settings, recent queries/search results, ...
  - Geographic locations:
    • Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, ...
- Scale is large
  - Billions of URLs, many versions/page (~20K/ version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data

BigTable Goals

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time

- Need to support:
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets

- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls
BigTable

- Distributed multi-level map
  - With an interesting data model
- 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

Background: Building Blocks

Building blocks:
- **Google File System (GFS)**: Raw storage
- **Scheduler**: schedules jobs onto machines
- **Lock service**: distributed lock manager
  - Also can reliably hold tiny files (100s of bytes) w/ high availability
- **MapReduce**: simplified large-scale data processing

BigTable uses of building blocks:
- **GFS**: stores persistent state
- **Scheduler**: schedules jobs involved in BigTable serving
- **Lock service**: master election, location bootstrapping
- **MapReduce**: often used to read/write BigTable data
Typical Cluster

- Cluster Scheduling Master
- Lock Service
- GFS Master

Machine 1
- User Task
- BigTable Server
- Single Task
- Scheduler Slave
- GFS Chunkserver
- Linux

Machine 2
- BigTable Server
- User Task
- Scheduler Slave
- GFS Chunkserver
- Linux

Machine 3
- BigTable Master
- Scheduler Slave
- GFS Chunkserver
- Linux

Basic Data Model

- Distributed multi-dimensional sparse map
  
  \((row, column, timestamp) \rightarrow \text{cell contents}\)

- Good match for most of our applications
Rows

• Name is an arbitrary string
  – Access to data in a row is atomic
  – Row creation is implicit upon storing data
• Rows ordered lexicographically
  – Rows close together lexicographically usually on one or a small number of machines

Tablets

• Large tables broken into tablets at row boundaries
  – Tablet holds 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:
    • 100 machines each pick up 1 tablet from failed machine
  – Fine-grained load balancing
    • Migrate tablets away from overloaded machine
    • Master makes load-balancing decisions
Tablets & Splitting

<table>
<thead>
<tr>
<th>aaa.com</th>
<th>cnn.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com/sports.html</td>
<td>TABLETS</td>
</tr>
<tr>
<td>Website.com</td>
<td>...</td>
</tr>
<tr>
<td>Zuppa.com/menu.html</td>
<td>...</td>
</tr>
</tbody>
</table>

“language”
“contents”

EN
"<html>..."

Tablets & Splitting

<table>
<thead>
<tr>
<th>aaa.com</th>
<th>cnn.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com/sports.html</td>
<td>TABLETS</td>
</tr>
<tr>
<td>Website.com</td>
<td>...</td>
</tr>
<tr>
<td>Yahoo.com/kids.html</td>
<td>...</td>
</tr>
<tr>
<td>Zuppa.com/menu.html</td>
<td>...</td>
</tr>
</tbody>
</table>

“language”
“contents”

EN
"<html>..."
System Structure

Bigtable cell

Bigtable master
performs metadata ops, load balancing

Bigtable client library

Bigtable tablet server
serves data

Bigtable tablet server
serves data

Bigtable tablet server
serves data

Cluster Scheduling Master
handles failover, monitoring

GFS
holds tablet data, logs

Lock service
holds metadata, handles master-election

Questions

The BigTable master are not replicated for correctness/availability. Why?

– Hint: It is replicated as a performance optimization

The tablet servers are not replicated for correctness/availability. Why?
Fault tolerance

• If a tablet server fails (while storing ~100 tablets)
  – reassign each tablet to another machine
  – so 100 machines pick up just 1 tablet each
  – tablet SSTables & log are in GFS

• If the master fails
  – acquire lock from Chubby to elect new master
  – read config data from Chubby
  – contact all tablet servers to ask what they’re responsible for

Is BigTable ACID?

• Durability and atomicity: via GFS
• Strong consistency: operations processed by a single server in order
• Isolated transactions within a single key
• Multi-key transactions added in Spanner
Locating Tablets

- 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
- Could use consistent hashing
  - Would spread related data across multiple tablets
- Could use the BigTable master
  - Central server would be bottleneck in large system
- Instead: store special tables containing tablet location info in BigTable cell itself

Locating Tablets (cont.)

- Our approach: 3-level hierarchical lookup scheme for tablets
  - Location is ip:port of relevant server
  - 1st level: bootstrapped from lock server, points to owner of META0
  - 2nd level: Uses META0 data to find owner of appropriate META1 tablet
  - 3rd level: META1 table holds locations of tablets of all other tables
    - META1 table itself can be split into multiple tablets

Figure 4: Tablet location hierarchy.
Tablet Representation

- SSTable: Immutable on-disk ordered map from string \( \rightarrow \) string
- String keys: \(<row, column, timestamp>\) triples

![Diagram of Tablet Representation]

Compactions

- Tablet state represented as set of immutable compacted SSTable files, plus tail of log (buffered in memory)

- Minor compaction:
  - When in-memory state fills up, pick tablet with most data and write contents to SSTables stored in GFS
    - Separate file for each locality group for each tablet

- Major compaction:
  - Periodically compact all SSTables for tablet into new base SSTable on GFS
    - Storage reclaimed from deletions at this point
Columns

- Columns have two-level name structure:
  - Family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional level of indexing, if desired

Timestamps

- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - "Return most recent K values"
  - "Return all values in timestamp range (or all values)"
- Column families can be marked w/ attributes:
  - "Only retain most recent K values in a cell"
  - "Keep values until they are older than K seconds"
API

• Metadata operations
  – Create/delete tables, column families, change metadata
• Writes (atomic)
  – 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

Shared Logs

• Designed for 1M tablets, 1000s of tablet servers
  – 1M logs being simultaneously written performs badly
• Solution: shared logs
  – Write log file per tablet server instead of per tablet
    • Updates for many tablets co-mingled in same file
  – Start new log chunks every so often (64MB)
• Problem: during recovery, server needs to read log data to apply mutations for a tablet
  – Lots of wasted I/O if lots of machines need to read data for many tablets from same log chunk
Shared Log Recovery

Recovery:
- Servers inform master of log chunks they need to read
- Master aggregates and orchestrates sorting of needed chunks
  - Assigns log chunks to be sorted to different tablet servers
  - Servers sort chunks by tablet, writes sorted data to local disk
- Other tablet servers ask master which servers have sorted chunks they need
- Tablet servers issue direct RPCs to peer tablet servers to read sorted data for its tablets

Compression

- Many opportunities for compression
  - Similar values in the same row/column at different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows

- Within each SSTable for a locality group, encode compressed blocks
  - Keep blocks small for random access (~64KB compressed data)
  - Exploit fact that many values very similar
  - Needs to be low CPU cost for encoding/decoding
Compression Effectiveness

- Experiment: store contents for 2.1B page crawl in BigTable instance
  - Key: URL of pages, with host-name portion reversed
  - Groups pages from same site together
    - Good for compression (neighboring rows tend to have similar contents)
    - Good for clients: efficient to scan over all pages on a web site
- One compression strategy: gzip each page: ~28% bytes remaining
- BigTable: BMDiff + Zippy

<table>
<thead>
<tr>
<th>Type</th>
<th>Count(B)</th>
<th>Space(TB)</th>
<th>Compressed</th>
<th>%remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web contents</td>
<td>2.1</td>
<td>45.1</td>
<td>4.2</td>
<td>9.2</td>
</tr>
<tr>
<td>Links</td>
<td>1.8</td>
<td>11.2</td>
<td>1.6</td>
<td>13.9</td>
</tr>
<tr>
<td>Anchors</td>
<td>126.3</td>
<td>22.8</td>
<td>2.9</td>
<td>12.7</td>
</tr>
</tbody>
</table>

Summary of BigTable Key Ideas

Unstructured key-value table data
  - No need for having a schema in advance
  - instead create columns when needed

Versioned data, with key-specific garbage collection

Maintain data locality on same tablet
  - Instead of consistent hashing, reconfigure tablet boundaries for load balancing

Tablets for lookup: key -> tablet

Efficient updates using log structure (store deltas)
BigTable in retrospect

- Definitely a useful, scalable system!
- Still in use at Google, motivated lots of NoSQL DBs
- Biggest mistake in design (per Jeff Dean, Google): not supporting distributed transactions!
  - became really important w/ incremental updates
  - users wanted them, implemented themselves, often incorrectly!

Megastore Motivation

- Many applications need transactions that span multiple rows
  - Examples: gmail, google+, picasa, ...
- Key-value store that spans multiple data centers
  - Fast local reads
  - At cost of slower writes
Megastore

- Replicate data using BigTable as underlying key-value store
  - BigTable copy per data center
- Two phase commit for multi-key transactions
  - Store 2pc log as “column” in BigTable
- Fast reads: in normal case, read lease provided to all data centers
- Slow writes: revoke read leases from all data centers before performing write