Spanner Motivation

Tom Anderson

Outline

Last week:
  – Chubby: coordination service
  – BigTable: scalable storage of structured data
  – GFS: large-scale storage for bulk data

Today/Friday:
  – Lessons from GFS/BigTable
  – Megastore: Multi-key, multi-data center NoSQL
  – Spanner: Multi-key, multi-data center NoSQL using real-time
• each file stored as 64MB chunks
• each chunk on 3+ chunkservers
• single master stores metadata

At Least Once Append

• If failure at primary or any replica, retry append (at new offset)
  – Append will eventually succeed!
  – May succeed multiple times!
• App client library responsible for
  – Detecting corrupted copies of appended records
  – Ignoring extra copies (during streaming reads)
• Why not append exactly once?
Question

Does the BigTable tablet server use “at least once append” for its operation log?

Data Corruption

• Files stored on Linux, and Linux has bugs
  – Sometimes silent corruptions
• Files stored on disk, and disks are not fail-stop
  – Stored blocks can become corrupted over time
  – Ex: writes to sectors on nearby tracks
  – Rare events become common at scale
• Chunkservers maintain per-chunk CRCs (64KB)
  – Local log of CRC updates
  – Verify CRCs before returning read data
  – Periodic revalidation to detect background failures
~15 years later

• Scale is much bigger:
  – now 10K servers instead of 1K
  – now 100 PB instead of 100 TB

• Bigger workload change: updates to small files!

• Around 2010: incremental updates of the Google search index

GFS -> Colossus

• GFS scaled to ~50 million files, ~10 PB
• Developers had to organize their apps around large append-only files (see BigTable)
• Latency-sensitive applications suffered
• GFS eventually replaced with a new design, Colossus
Metadata scalability

- Main scalability limit: single master stores all metadata
- HDFS has same problem (single NameNode)
- Approach: partition the metadata among multiple masters
- New system supports ~100M files per master and smaller chunk sizes: 1MB instead of 64MB

Reducing Storage Overhead

- Replication: 3x storage to handle two copies
- Erasure coding more flexible: m pieces, n check pieces
  - e.g., RAID-5: 2 disks, 1 parity disk (XOR of other two) => 1 failure w/ only 1.5 storage
- Sub-chunk writes more expensive (read-modify-write)
- Recovery is harder: usually need to get all the other pieces, generate another one after the failure
**Erasure Coding**

- 3-way replication:
  3x overhead, 2 failures tolerated, easy recovery
- Google Colossus: (6,3) Reed-Solomon code
  1.5x overhead, 3 failures
- Facebook HDFS: (10,4) Reed-Solomon
  1.4x overhead, 4 failures, expensive recovery
- Azure: more advanced code (12, 4)
  1.33x, 4 failures, same recovery cost as Colossus

---

**BigTable System Structure**
Tablet Representation

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

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
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
• Lack of distributed transactions: biggest mistake in design, per Jeff Dean
• Lack of multi-data center support
Question

How would you add multi-key transactions to BigTable?

- Easy if all keys are on the same tablet, or on different tablets on the same tablet server
- What if keys are on different tablet servers?

Multi-Key NoSQL Transactions

- Straw proposal: Two phase commit
  - Select one tablet server as coordinator
  - Add log entries for coordinator/participant actions
  - Check log if coordinator or participant fails
- What if coordinator/participant crashes?
  - BigTable master wait for lease timeout
  - Select new tablet server
  - New tablet server recovers in progress transactions using log
  - Abort/commit as appropriate
Performance of NoSQL 2PC

What is performance of multi-key transactions using two phase commit?
- Each tablet server orders operations to its own keys
- If coordinator, must lock or delay subsequent operations to that key, until participants reply
- If participant, must lock or delay subsequent ops to that key, until coordinator commits/aborts
- All ops to key are delayed, not just multi-key ones
- Stale reads to the rescue?

Question

How would you add support for multiple data centers to BigTable?
Multiple Data Center NoSQL

- Straw proposal: Paxos state machine replicas
  - Every data center has complete copy of data
  - One serves as Paxos leader (per tablet or per key)
  - Clients contact leader
  - Leader proposes ordering of client ops to tablet

- Paxos implies
  - correct despite data center failures, network partitions, etc.
  - Progress if a majority of data centers remain up and connected

Multi-Data Center Paxos Performance

- Assume Paxos is optimized: one round from leader to participants per operation (batched)

- Latency:
  - One RT from client to (remote) leader
  - One RT from leader to farthest data center

- Throughput
  - Every operation sent to every data center
  - N messages to coordinate Paxos ordering (batched)

- Per-transaction: two log operations at coordinator, two at each participant

- Stale reads to the rescue?
Megastore Motivation

• Many apps need atomicity across rows
  – Examples: gmail, google+, picasa, ...
• Many apps need to span multiple data centers
  – Hide outages from end users
  – Low latency for every user on planet
• Goals:
  – Fast local reads
  – At cost of slower writes

Megastore Key Ideas

• BigTable as a service
  – No need to reimplement NoSQL
  – Two phase commit across keys
  – operation log stored in a BT column
• Use data center for testing
  – Extensive randomized testing of corner cases
Megastore Key Ideas

• Paxos with replica in each data center
  – Most operations are reads
  – For writes, rotating leader – wait turn to propose
• Special quorum rules
  – reads require one replica, can always be local
  – Writes require majority (of data centers) to commit
  – Writes require all replicas before return to client
  – If data center fails, wait for lease expire, then return to client