Bigtable: A Distributed Storage System for Structured Data

Huijie Pan, Ray Hung, Akhila, David Liu, Nandha Sundaravadivel

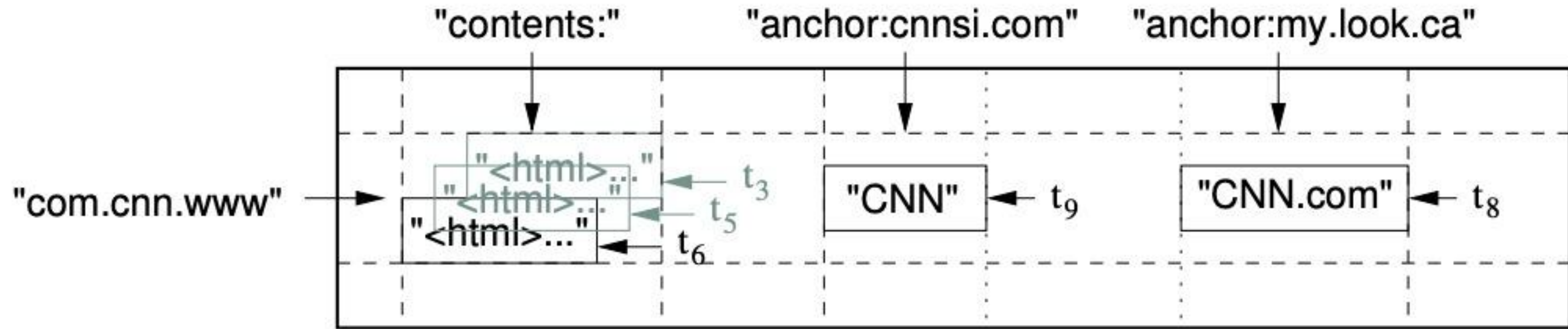
What is Bigtable?

- Distributed storage system for managing structured data at massive scale
- Designed to scale to petabytes of data across thousands of servers
- Key features:
  - Handles structured data
  - Wide applicability (web indexing to Google Earth)
  - High performance and availability
  - Simple data model

Why Bigtable?

- Problem being solved:
  - Need to handle diverse workloads
  - Must scale horizontally
  - Require high performance for both:
    - Large sequential reads/writes
    - Random reads/writes
- Real usage examples:
  - Google Analytics
  - Google Earth
  - Personalized Search

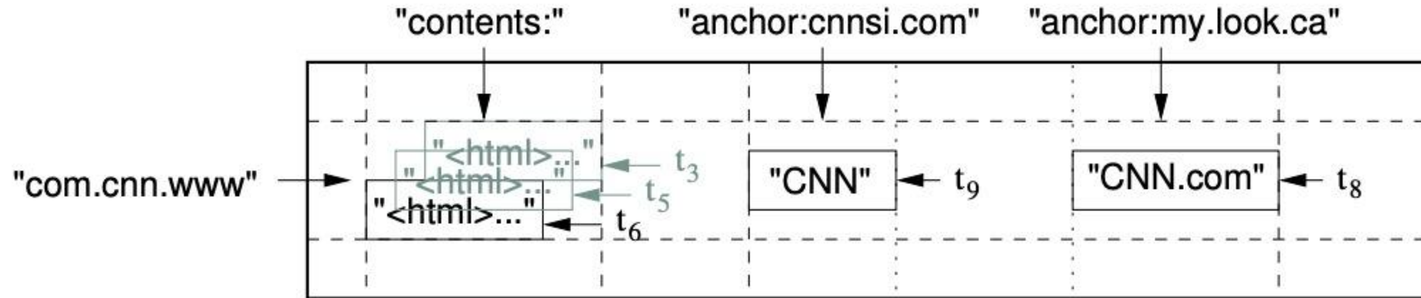
Data Model Overview



- Key points:
  - Think of it as a giant sorted map
  - Data is indexed by: (row key, column key, timestamp) → value
  - Real example: Storing web pages and their references

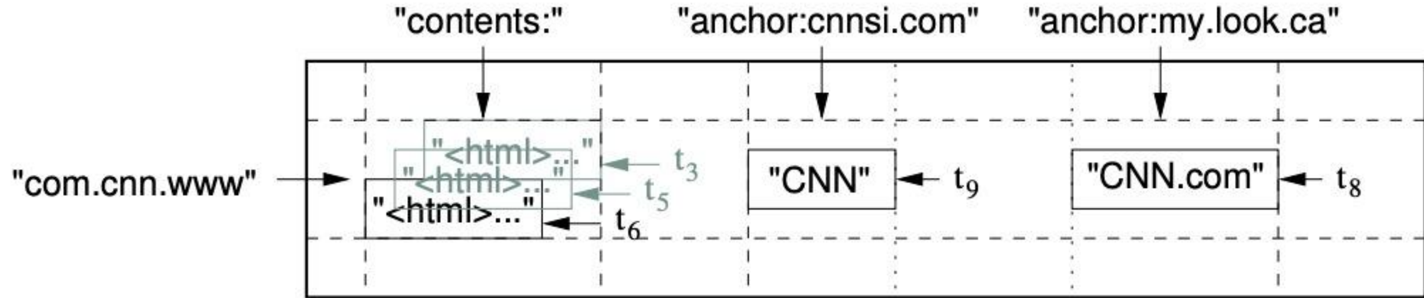
Example is from [Bigtable: A Distributed Storage System for Structured Data]

Row Keys: The First Dimension



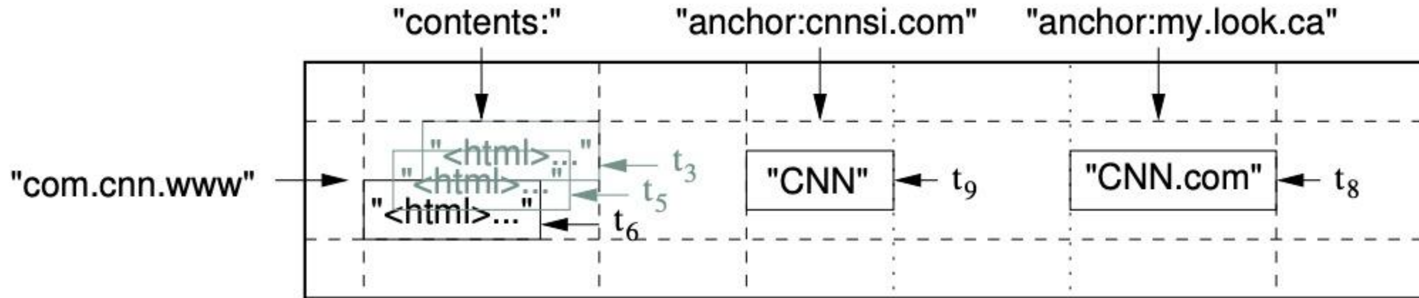
- Row key: "com.cnn.www" (reversed URL)
- Rows are sorted lexicographically
- Why reverse URLs?
  - Domains like com.cnn.* are grouped together
  - Enables efficient domain-specific queries
- Row keys enable range scans
- Each row is dynamically partitioned into tablets

Column Families: Grouping Related Data



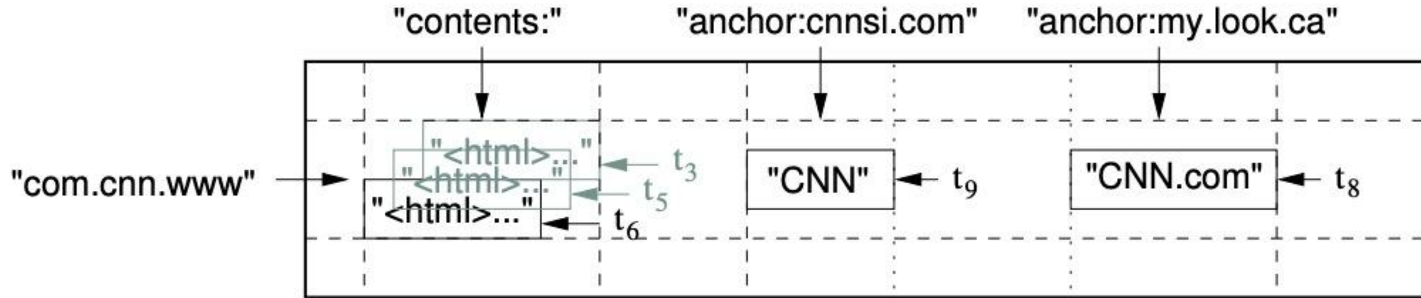
- Two column families in example:
  - "contents:" - stores actual web page content
  - "anchor:" - stores incoming links
- Key Properties:
  - Must be created before storing data
  - Access control at family level
  - Same family typically contains similar type of data
  - Example: All anchors stored in "anchor:" family
  - Different families can have different compression settings

Column Qualifiers: Dynamic Columns



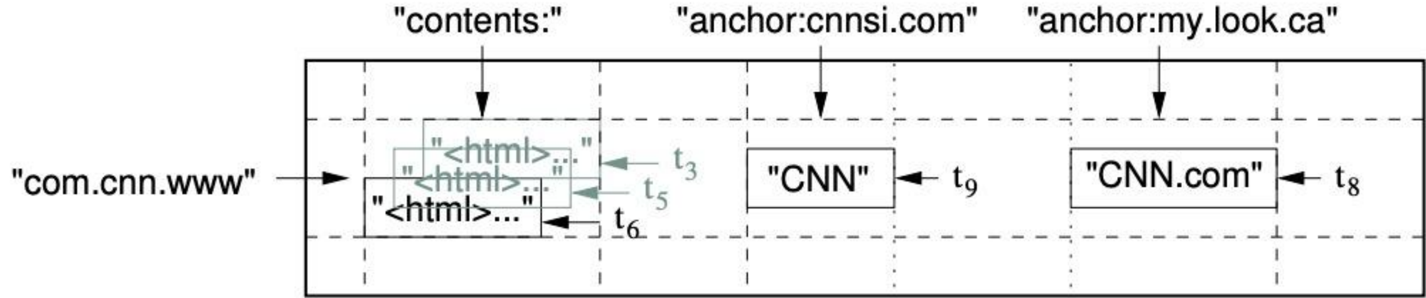
- In the example:
  - "contents:" has qualifier "html"
  - "anchor:" has qualifiers "cnnsi.com" and "my.look.ca"
- Key points:
  - Qualifiers can be created on the fly
  - Can have huge number of columns
  - Format: family:qualifier
  - Enables flexible schema
  - Example: Each linking site gets its own qualifier

Timestamps: Built-in Versioning



- From example:
  - Contents has versions at t3, t5, t6
  - Anchors have single versions at t8, t9
- Key features:
  - Multiple versions of same cell
  - Automatic garbage collection
  - Can be set by client or system
  - Enables time-travel queries
  - Configurable version retention

Data Model Summarize:



So if you want to:

- Find all CNN pages: Scan rows starting with "com.cnn"
- Get latest version of CNN homepage: Look up (com.cnn.www, contents:html) with latest timestamp
- See who links to CNN: Look at all qualifiers in the anchor family
- See how CNN's page changed: Look at different timestamps of contents:html
- Find when SI linked to CNN: Check timestamp t9 of anchor:cnnsi.com

Data Model Benefits

This Model is powerful because

- It's sparse - you don't waste space on empty cells
- It's flexible - new columns (qualifiers) can be added anytime
- It's versioned - you can track changes over time
- It's grouped logically - related data stays together
- It's efficient for both random access and scans

BigTable API

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

Figure from [Bigtable: A Distributed Storage System for Structured Data]

BigTable API

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

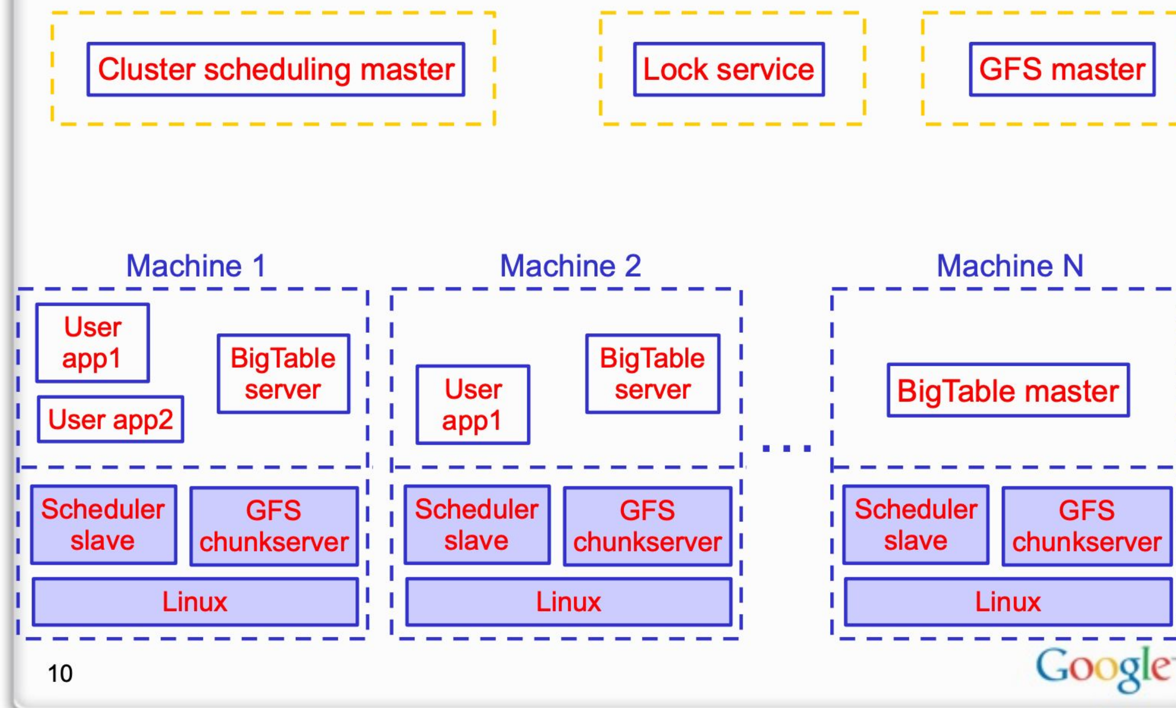
Figure 3: Reading from Bigtable.

Figure from Bigtable: A Distributed Storage System for Structured Data

BigTable Building Blocks

- Google File System (GFS)
  - Store persistent state such as log and data files
- Scheduler
  - schedules jobs involved in BigTable serving
- Distributed Lock service - Chubby
  - master election, location bootstrapping, discover tablet servers, store BigTable schema and access control lists
- MapReduce
  - BigTable can be input and/or output for MapReduce computations

Slide adapted from BigTable: A System for Distributed Structured Storage By Jeff Dean



Took from BigTable: A System for Distributed Structured Storage By Jeff Dean

Google SSTable File Format

- used internally to store Bigtable data

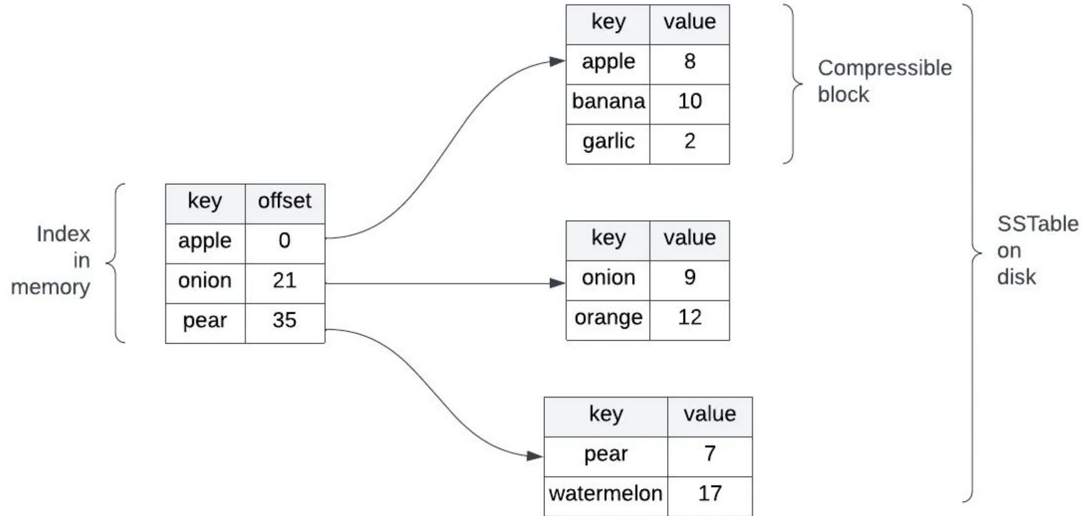


Diagram from [SSTable database or big ideas behind the scene]

Distributed Lock Service - Chubby

- Five active replicas, one of which is master
- Uses the Paxos algorithm to keep its replicas consistent
- Provide namespace that can be used as lock with atomic read/write
- Each Chubby client maintains a session with a Chubby service.
- BigTable uses Chubby for
  - master election
  - location bootstrapping
  - discover tablet servers and finalize tablet server deaths
  - store BigTable schema and access control lists

Implementation - Tablet location

Three-Level Hierarchy for efficient tablet location:

- Chubby file (root location).
- Root tablet (stores locations of METADATA tablets).
- METADATA tablets (store user tablet locations).

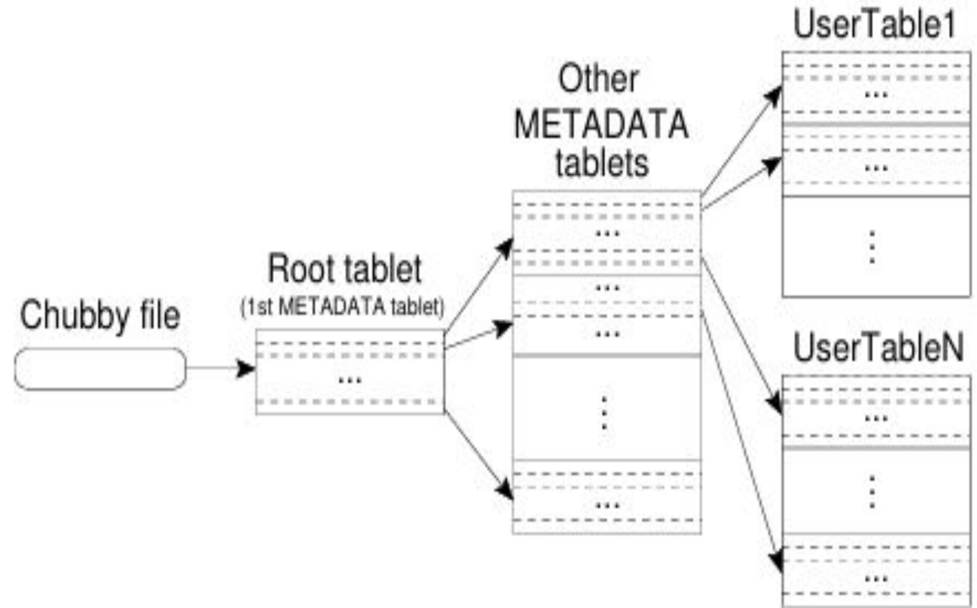
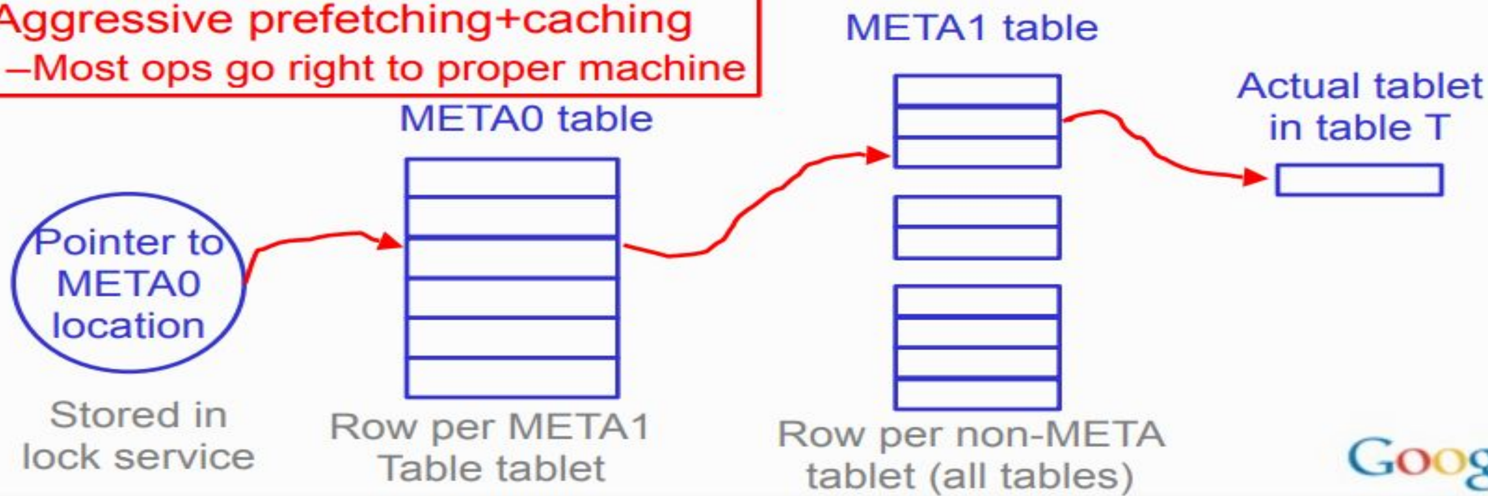


Figure 4: Tablet location hierarchy.

Locating Tablets (cont.)

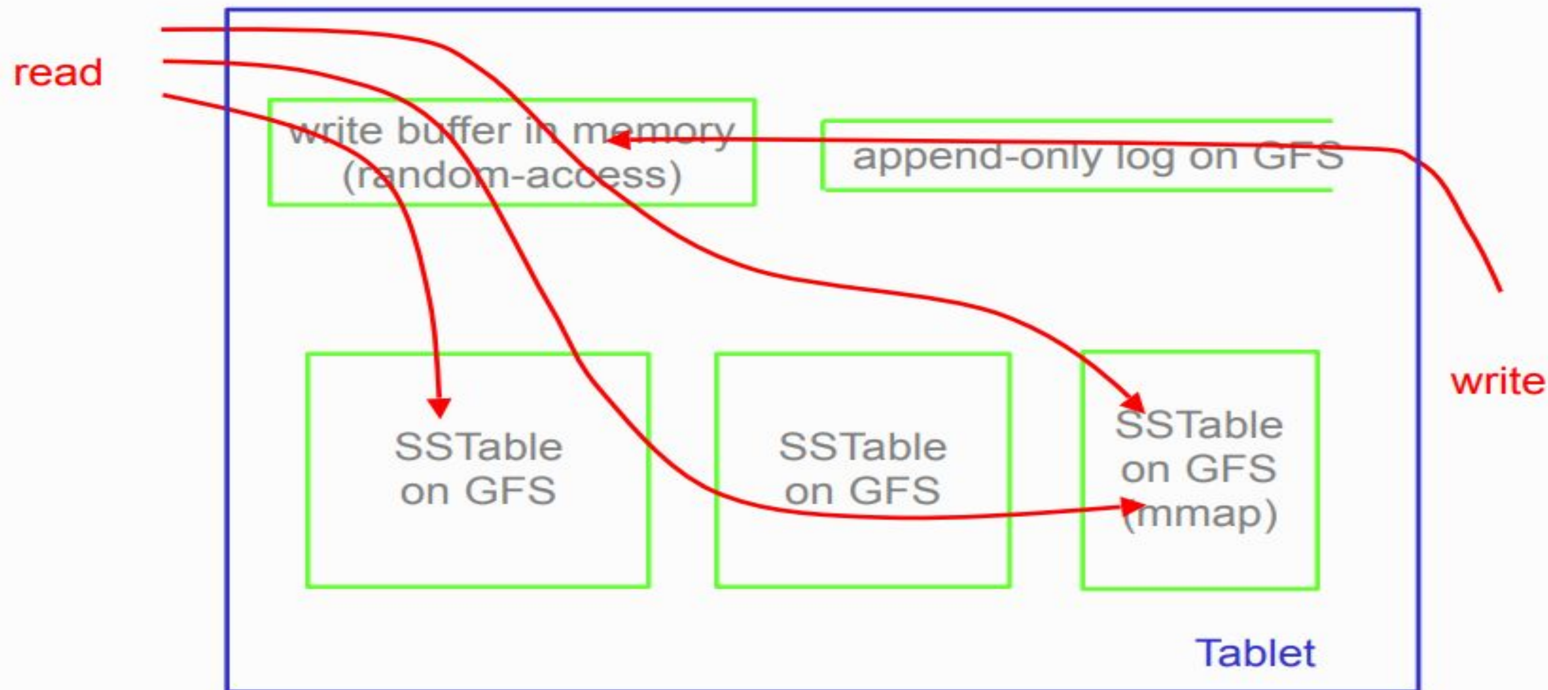
- Our approach: 3-level hierarchical lookup scheme for tablets
  - Location is *ip:port* of relevant server
  - 1st level: bootstrapped from lock service, points to owner of META0
  - 2nd level: Uses META0 data to find owner of appropriate META1 tablet
  - 3rd level: META1 table holds locations of tablets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



Implementation - Tablet Assignment

- Role of the Master Server
  - Monitor live Tablet Servers
  - Track active servers and manage tablets
  - Load balancing
- Tablet Server Registration with Chubby
  - Unique lock to register tablet server
  - On the event of lock loss, stops serving tablets.
- Detecting and Handling Server Failures
  - Periodic polling on the servers by master
  - Master acquires failed server’s lock -> Deletes -> Moves impacted tablets to unassigned pool
- Reassigning Unassigned Tablets
  - Master sends tablet load request to assign a server to tablets in the pool
  - Maintain high availability
  - Minimally disruptive during reassignment

Tablet Representation



SSTable: Immutable on-disk ordered map from string->string
string keys: *<row, column, timestamp>* triples

Google

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

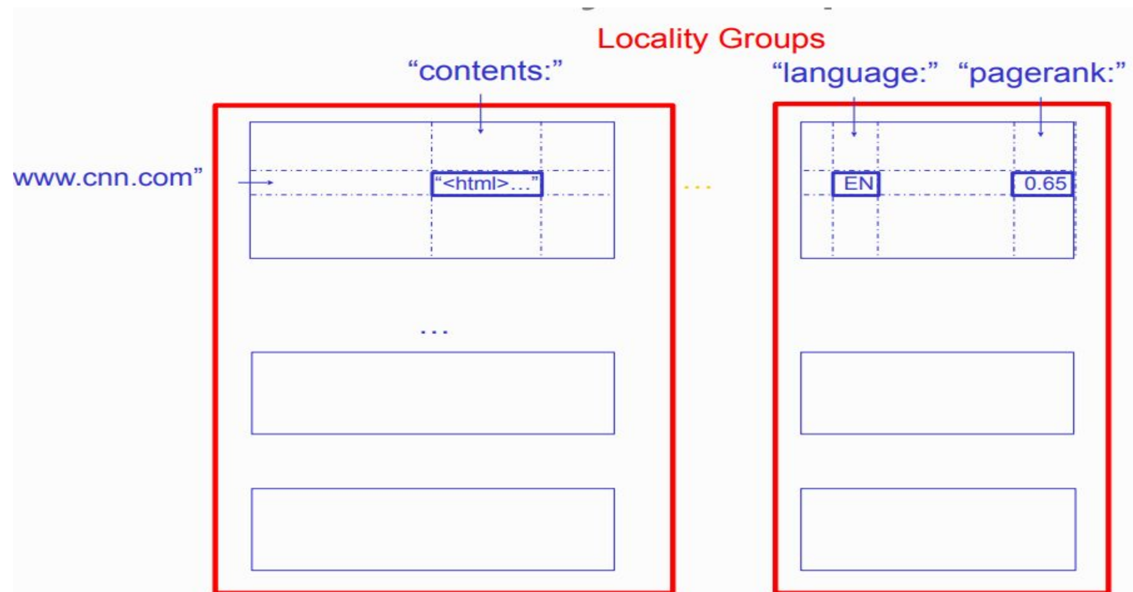
- Major compaction:

– Periodically compact all SSTables for tablet into new base SSTable on GFS

    - Storage reclaimed from deletions at this point

Refinements - Locality Groups

- Column families can be grouped into locality groups
- **Example**: The METADATA table uses an in-memory locality group for the location column family



Refinements - Compression

- Compression can be enabled or disabled per locality group
- Compression is applied to individual SSTable blocks
- Google uses a custom two-pass compression scheme:
    - Bentley and McIlroy algorithm for compressing long common strings across a large window
    - a fast compression algorithm for compact storage
- This compression approach achieves high speeds:
    - **Encoding**: 100-200 MB/s
    - **Decoding**: 400-1000 MB/s
- Achieves high compression ratios; e.g., 10-to-1 reduction in Webtable data.
- Designed for speed without sacrificing significant space savings.

Refinements - Contd.,

Caching for Read Performance

- Scan Cache: For frequently accessed key-value pairs.
- Block Cache: For blocks of data near recently read data.

Bloom Filters

- Reduce disk accesses for read operations
- Check if SSTable might contain data for a row/column pair

Commit-Log Implementation

- One commit log per tablet server.
- Combines group commit optimization
- Parallel recovery process sorts logs by tablet to reduce read time on recovery

Exploiting Immutability.

- Immutable SSTables simplify data access and concurrency control.
- Enables quick tablet splits by allowing child tablets to share parent SSTables.

Evaluation

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

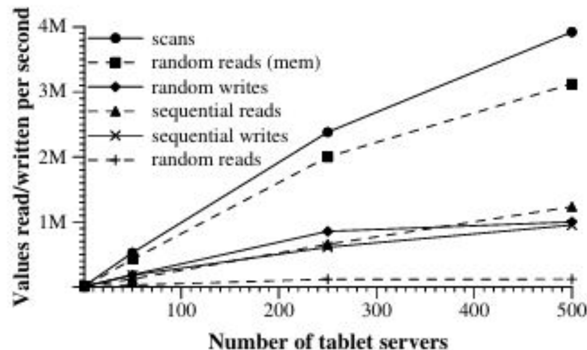


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

- Write Benchmark
  - Sequential: row keys are stored in contiguous manner from 0 to R-1
  - Random: row keys were generated and hashed to distribute across key space
- Read benchmark
  - Sequential/Random: rows are accessed sequentially/randomly
  - Mem: data in the benchmark is served from memory instead of from disk
- Scan
  - Read using BigTable API to scan all values in a row range
  - Reduce the number of RPC

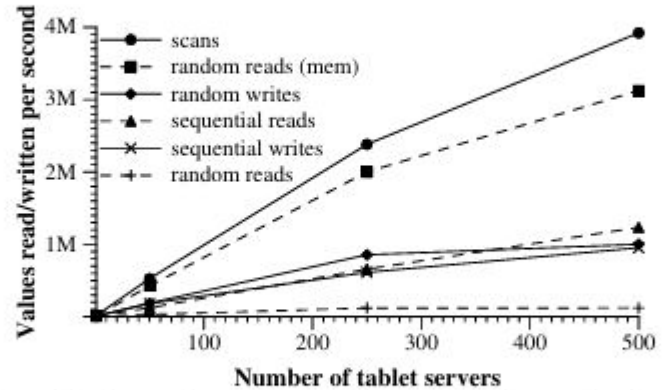
Observations

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.

- Read Performance
  - Random read is the slowest since it often require fetching a 64KB block for each 1KB read
  - Reading from memory reduces the overhead of fetching 64KB block from GFS
  - Sequential read allows for caching of large data blocks
- Write Performance
  - Both sequential and random writes performs better than reads since we append all writes to a single commit log and stream the writes as a group to GFS
- Scans
  - Really fast since server can return a large number of values for a single RPC

Scaling



- Increase in performance as we increase the number of tablet servers
  - Not linearly, limited by network and load balancing constraints
- Random read has the worst scaling due to the overhead of reading 64KB block for every 1 KB read

Applications

- Google Analytics
  - Site tracking reports like the number of unique visitors each day or page view per URL each day
  - Utilizes Bigtable to store Raw Click Table (200 TB), the row key is a tuple of website name and session start time
  - Summary Table (20 TB) - each summaries are generated by periodic mapreduce jobs from raw click table
- Google Earth
  - Store preprocessing table (70 TB) with raw satellite image and related data for preprocessing
    - Use batch processing to clean and consolidate the image into final serving data
  - Serving Table (500 GB) indexed preprocessed data in GFS
    - Hosted across hundred of tablet servers for high throughput and low latency (> 10,000 query / second)
- Personalized Search
  - User data table: stores each user's data in big table, identified by user id
    - Store all user interactions in the columns
    - Generate user profiles using mapreduce and use it to personalized live search results

Lessons

- Distributed Systems can Fail in the Craziest Ways
  - I.e. Clock skews, memory/network corruptions, hardware maintenance
- Clear Feature Scope
- Simplicity is Key
  - Code Simplicity
  - Dependency Simplicity

Related Works

- The Boxwood Project: Software Infrastructure for Large Scale Datastores
  - MacCormik et al., 2004
- Various work on distributed file systems
  - Can (Ratnasamy et al., 2001), Chord (Stoica et al., 2001), Tapestry (Zhao et al., 2001), and Pastry (Rowstron et al., 2001)
- Similar Large-Scale Industry Parallel Databases
  - Oracle Real Application Cluster DB (Oracle.com), IBM DB2 Parallel Edition (Baru et al., 1995)

Recent Developments

- Available on Google Cloud for external clients
- Multi Region Distribution
- Data Model Flexibility
    - JSON, Images, etc.
- (cloud.google.com)

Conclusions

- Bigtable: A distributed system for storing structured data at Google
    - Performance and high availability at scale
- Production use since 2005, 60 production users a year later
    - Still used today internally and available on Google Cloud Platform
- Further Questions/Comments
    - Cost to serve?

Study Question 1

Consider a system storing social media posts where each post has multiple comments, likes, and shares. Using Bigtable's data model, design a schema that would efficiently store this data. Explain:

- How would you structure the row keys to enable efficient queries for a user's posts?
- What column families would you create and why?
- How would you use column qualifiers to handle comments and likes?
- How would timestamps be useful in this scenario?

Compare your design choices with the web page example from the paper, explaining the similarities and differences in your approach.

Study Question 2

Imagine you're designing a system to store and manage all videos and comments for a video-sharing platform like YouTube. Consider Bigtable's key design decisions:

- Using a distributed architecture with a single master and multiple tablet servers
- Creating a sorted, distributed, persistent multidimensional map
- Providing a simple data model instead of a full relational model
- Using a distributed file system as the storage layer

For each of these choices:

- Explain what problem it solves in the context of video sharing
- Discuss what trade-offs it introduces
- Analyze how it impacts scalability and performance