# Advanced Cloud Computing

## BigTable

Wei Wang
CSE@HKUST
Spring 2025

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

# How should the semi-structured data be stored at a cloud scale?

Source: Google

# Outline

## Motivation and goals for BigTable

Data model

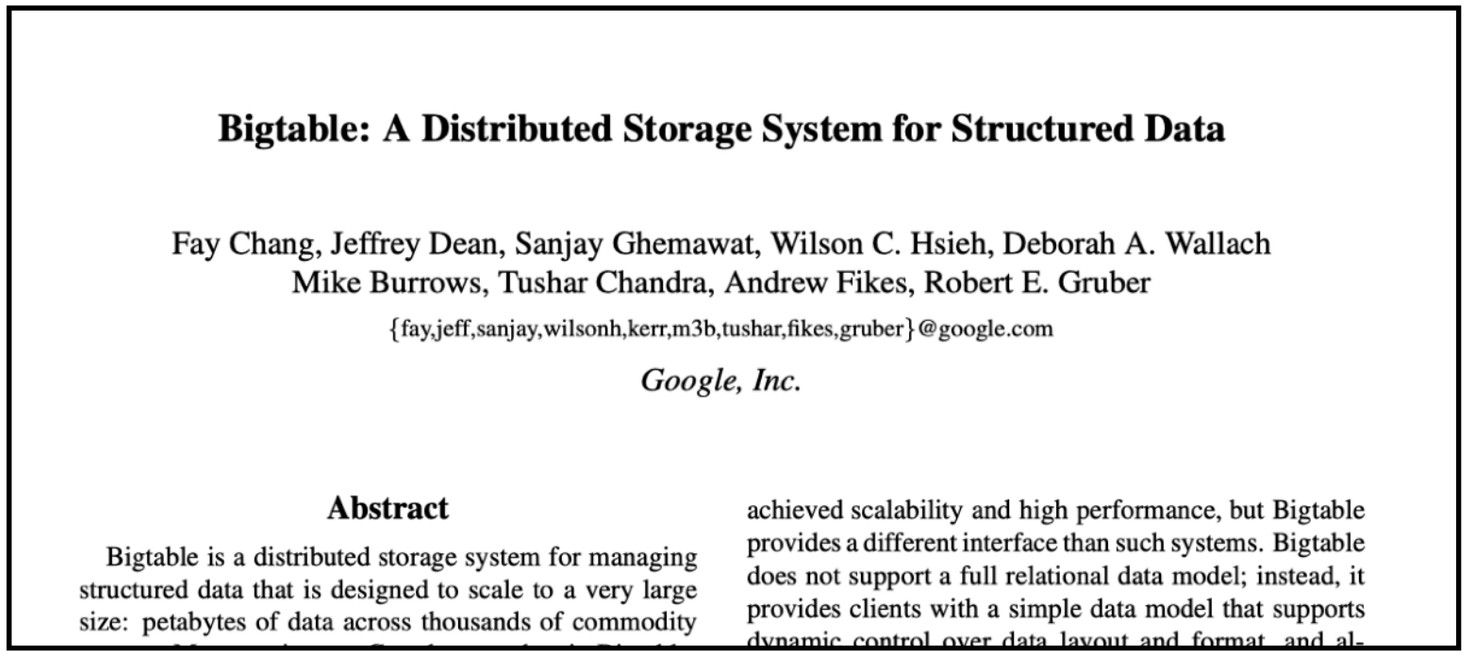
API

Building blocks

Implementation

Performance

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# Why build BigTable?

**Lots of (semi-)structured data at Google**

- URLs: contents, crawl metadata, links, anchors, pagerank, etc.
- Per-user data: user pref. settings, recent queries/search results, etc.
- Geographic locations: physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, etc.

**Scale is large**

- Billions of URLs, many versions/page (~20K/version)
- Hundreds of millions of users, thousands of queries per sec.
- 100TB+ of satellite image data

# Why not just use commercial DB?

**Scale is too large** for most commercial DB

Even if it weren't, cost would be very high

- Building internally means system can be applied across many projects for low incremental cost

Low-level storage optimizations help performance significantly

- Much harder to do when running on top of a database layer

# 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 (multiple versions)

**Self-managing**

- Servers can be added/removed dynamically
- Servers adjust to load balance

# Outline

Motivation and goals for BigTable

**Data model**

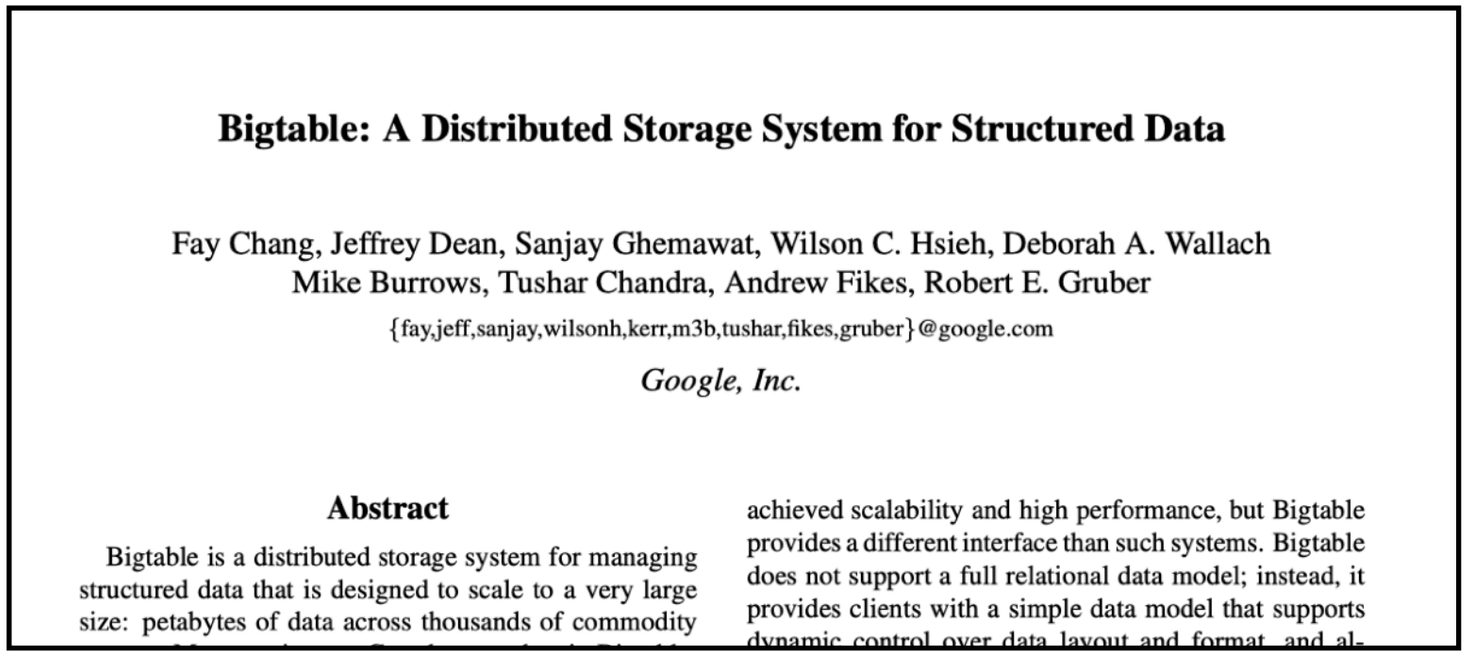
API

Building blocks

Implementation

Performance

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

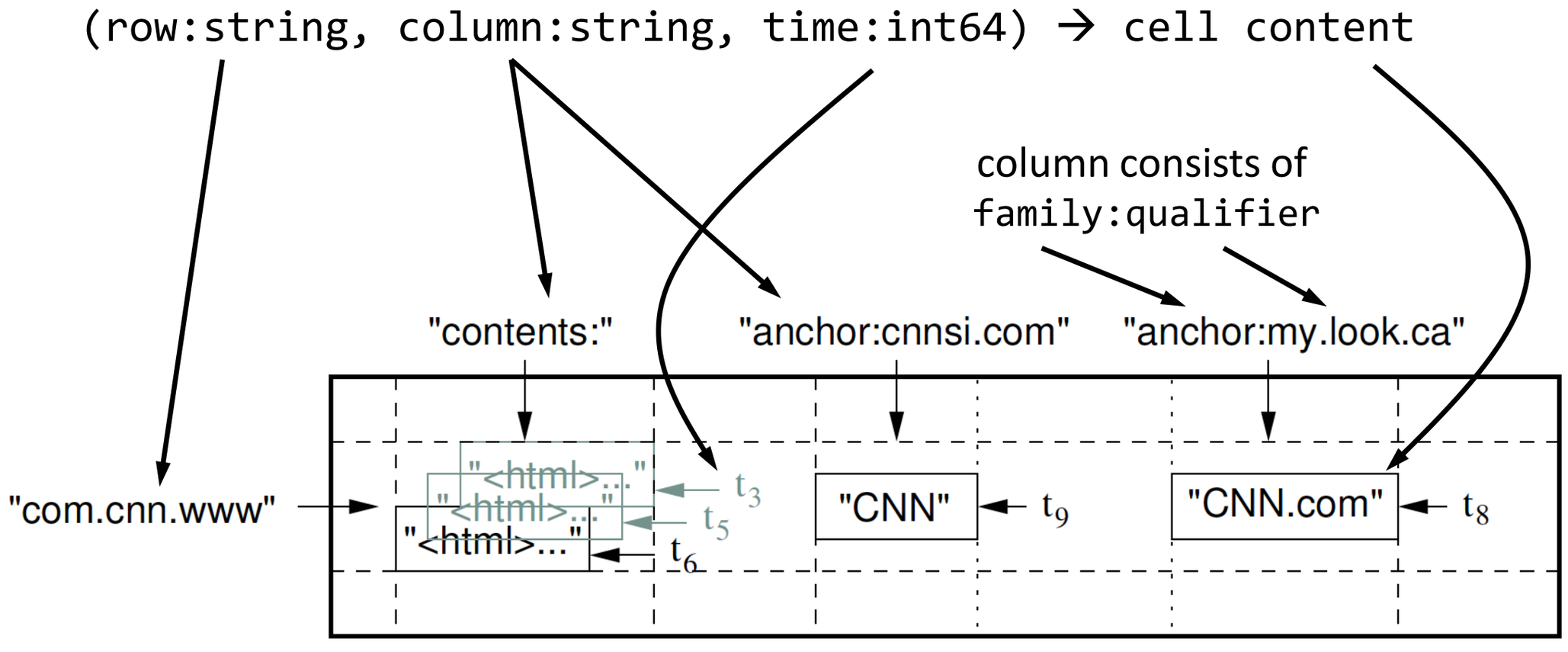
Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# BigTable and its data model

BigTable: a **distributed multi-level** sorted map that is **persistent, fault-tolerant, scalable,** and **self-managing**




BigTable's data model

# Rows

Row identified by **row key**

- 64KB in size (10-100 bytes typically used):
  - “com.cnn/www/downloads.html:ftp”
  - “com.cnn/www/index.html:http”
  - “com.cnn/www/login.html:https”

BigTable maintains data in **lexicographic order of row key**

- Rows lexicographically close usually on one or small number of machines

Row creation is implicit upon storing data

R/W under a single row key is **atomic**

No transactional support across rows

# Tablets

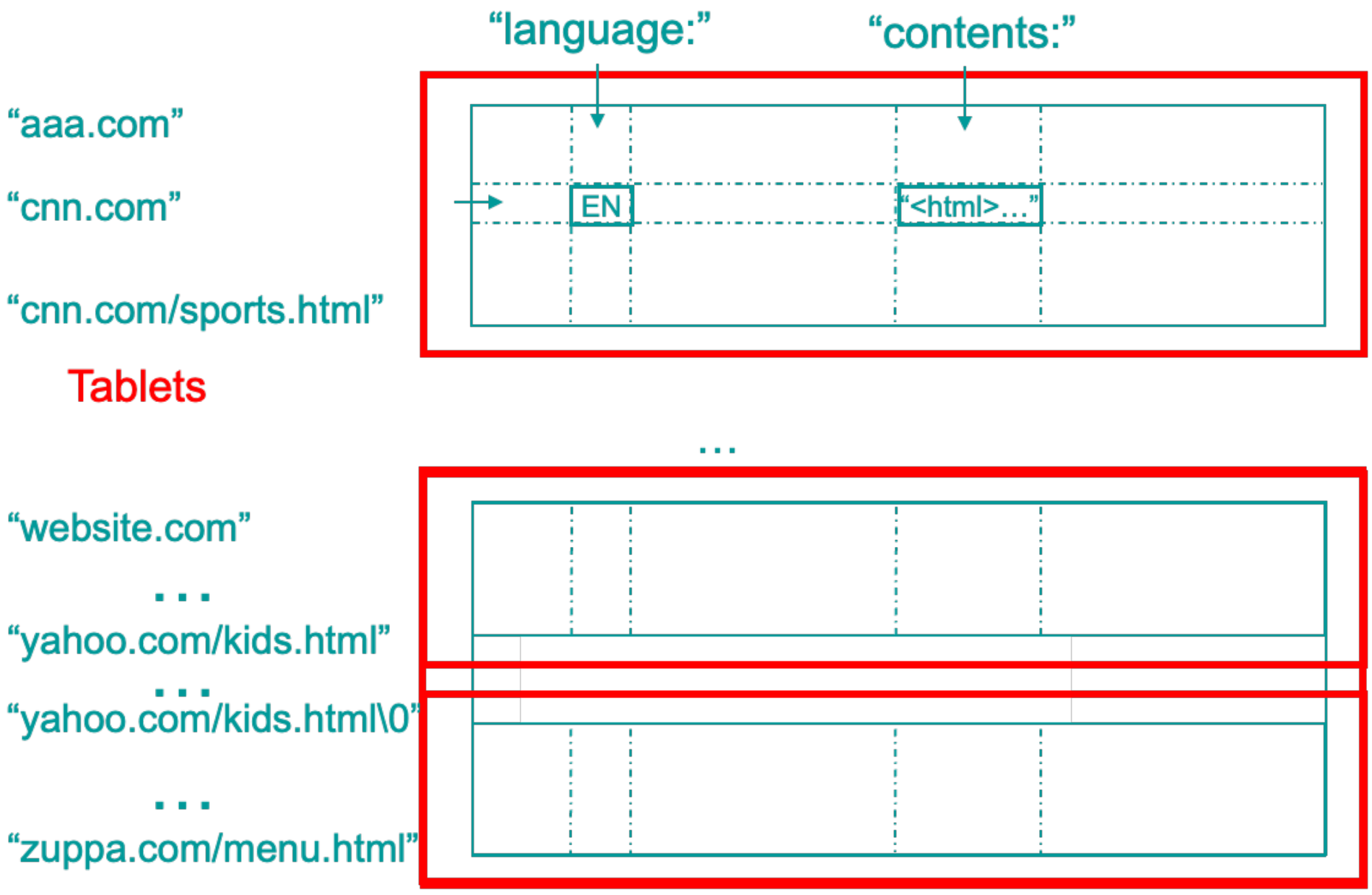
Large tables broken into **tablets** at row boundaries

- Tablet holds contiguous range of rows
  - Clients can often choose row keys to achieve locality
  - Aim for ~100MB to 200MB of data per table

Serving machine (table server) responsible for ~100 tablets

- Fast recovery
  - 100 machines each pic up 1 tablet from failed machine
- Fine-grained load balancing
  - Migrate tablets away from overloaded machine
  - Master makes load-balancing decisions

# Tablets & Splitting



# Columns

Columns have two-level name structure:

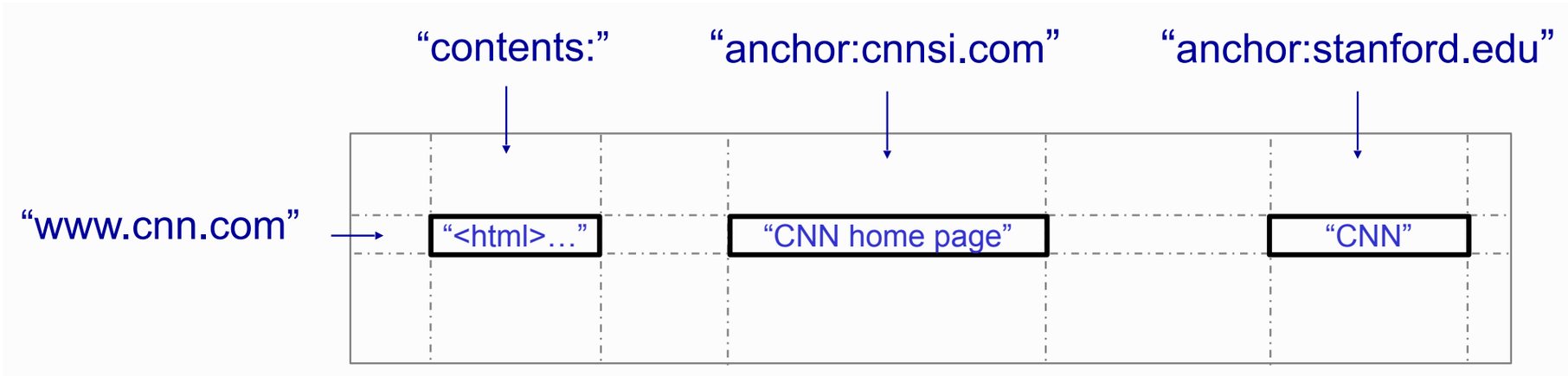
- **family:qualifier**

**Column family** is a group of column keys

- Specified on creation; basic unit of access control
- Same type of data in a family (compressed together)

**Qualifier** (column keys, or traditional columns)

- Can be created anytime, enabling unbounded columns



# Columns

Columns have two-level name structure:

- **family:qualifier**

You can think of each
(row, family) as a KV store:
`(qualifier, time) -> value`

row keys

column families

sorted rows

| | anchor | contents | language |
|---|---|---|---|
| ca.mylook | | | |
| com.cnn.www | cnnsi.com, $t_4$: CNN<br>cnnsi.com, $t_2$: CNN<br>mylook.ca, $t_1$: CNN.com | $t_6$: <html>…<br>$t_5$: <html>…<br>$t_3$: <html>… | EN |
| com.cnn.www/ca | | | |
| com.cnnsi.com | | | |

# Locality Groups

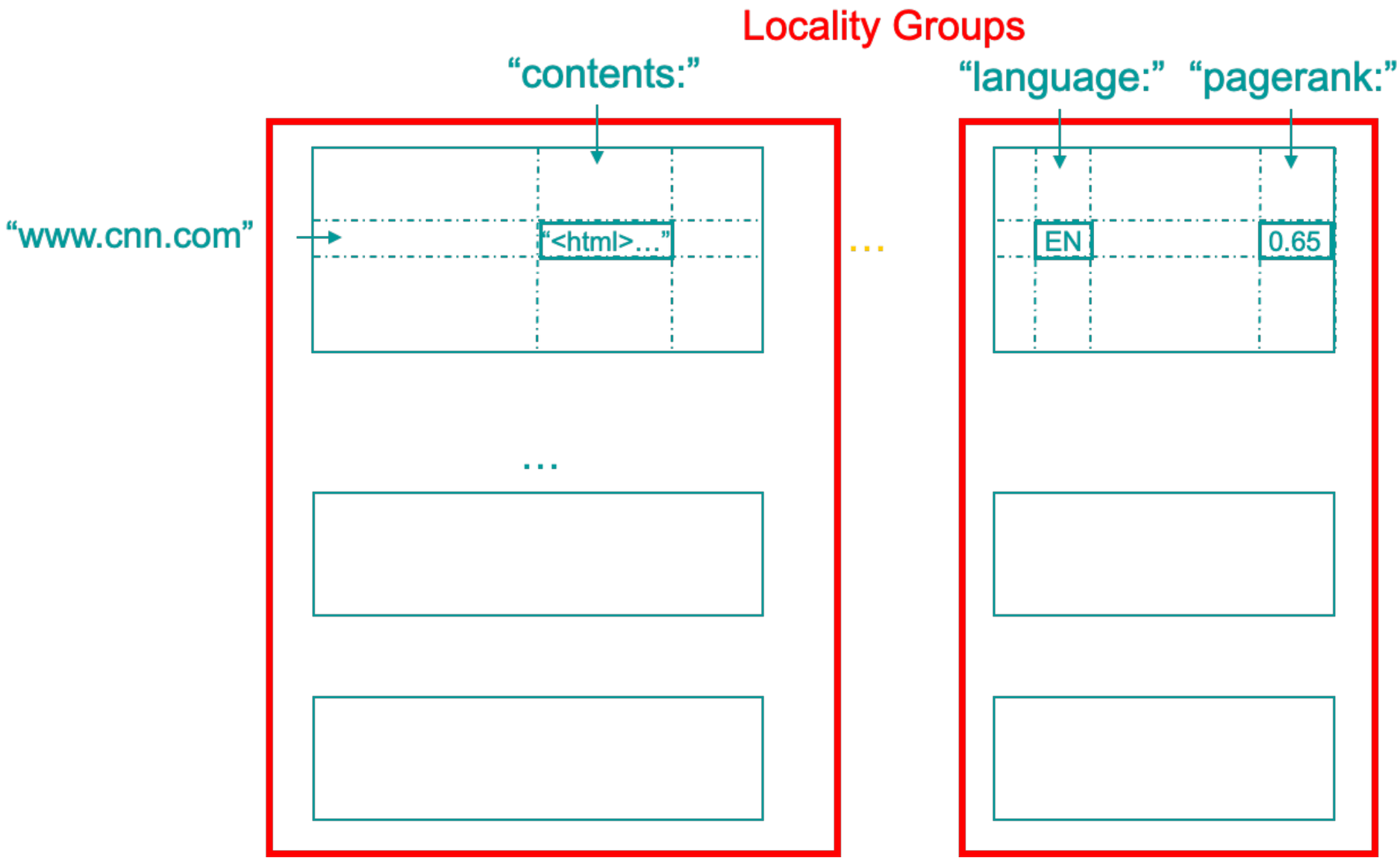
Column families can be assigned to a **locality group**

- Vertically split a table into multiple partitions
- Used to organize underlying storage representation for performance
  - scans over one locality group are ***O(bytes_in_locality_group)***, not ***O(bytes_in_table)***

Data in a locality group can be **explicitly memory-mapped**

- Caching to avoid frequent I/O

# Locality Groups




- Vertically split a table into multiple partitions

# Timestamps

Each cell can contain multiple versions of same data

- Version indexed by a 64-bit timestamp
- Real-time or assigned by client
- Data in decreasing order of timestamps

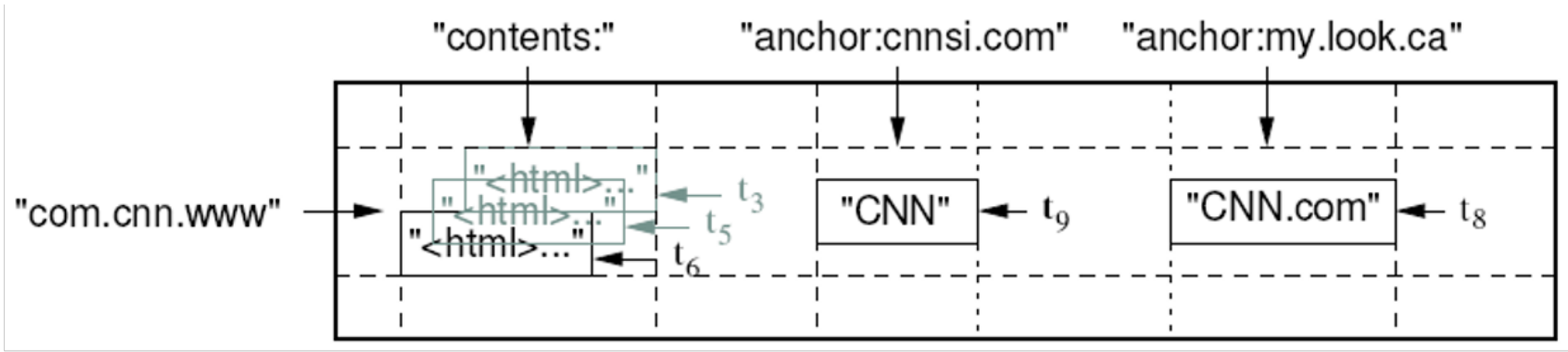
Lookup options

- “Return most recent *K* versions”
- “Return all versions in a timestamp range”

Per-column-family settings for garbage collection

- Keep only latest ***n*** versions
- Or keep only versions written since time ***t***

# Logical View & Physical Storage




"contents:" and "anchor:" are in different locality groups

| Row Key | Time Stamp | Column "contents:" |
|---|---|---|
| "com.cnn.www" | t6 | "<html>…" |
| | t5 | "<html>…" |
| | t3 | "<html>…" |

("com.cnn.www", "contents", t6)

| Row Key | Time Stamp | Column "anchor:" | |
|---|---|---|---|
| "com.cnn.www" | t9 | "anchor: cnnsi.com" | "CNN" |
| | t8 | "anchor: my.look.ca" | "CNN.com" |

("com.cnn.www", "anchor:my.look.ca", t8)

# Outline

Motivation and goals for BigTable

Data model

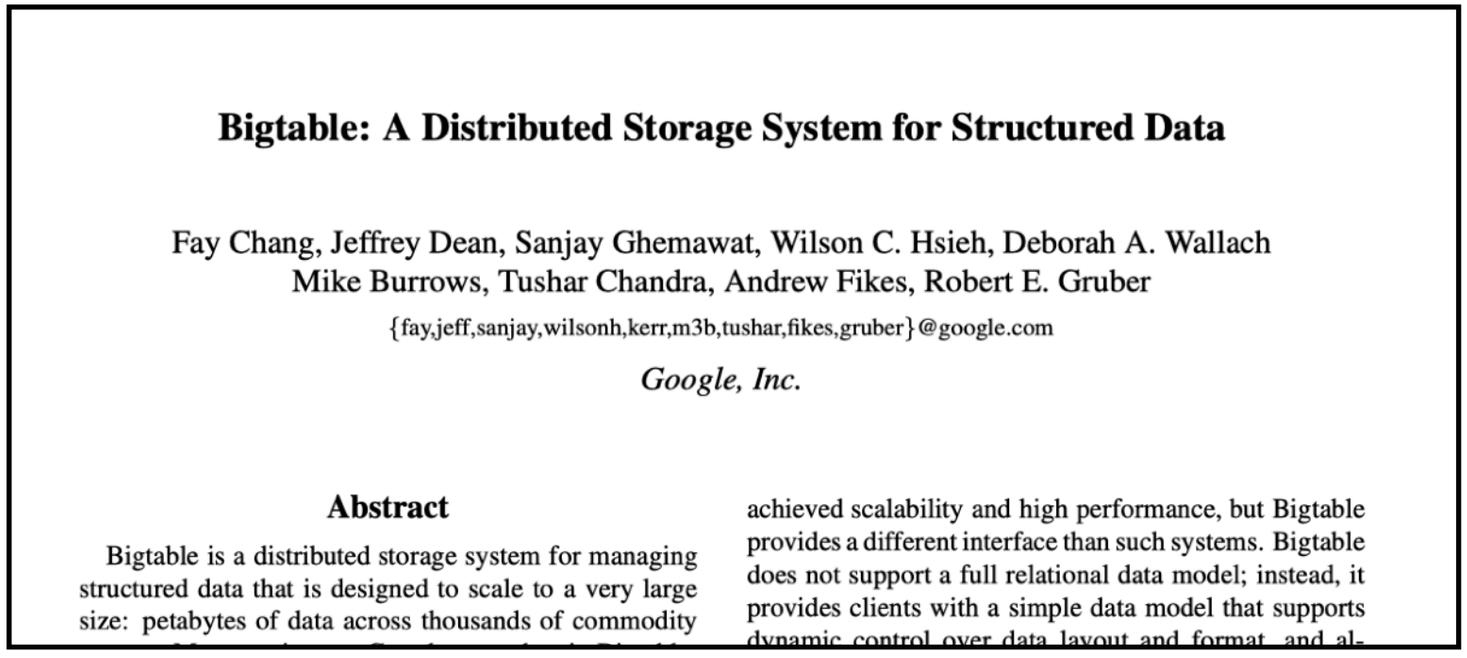
**API**

Building blocks

Implementation

Performance

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# BigTable API

Tables and column families

- Create, delete, update, control rights

Writes (**atomic**)

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

```
// 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); // ATOMIC
```

# BigTable API

## 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 one row, all rows, etc.
  - Can ask for all columns, just retain column families, or specific columns

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

# Outline

Motivation and goals for BigTable

Data model

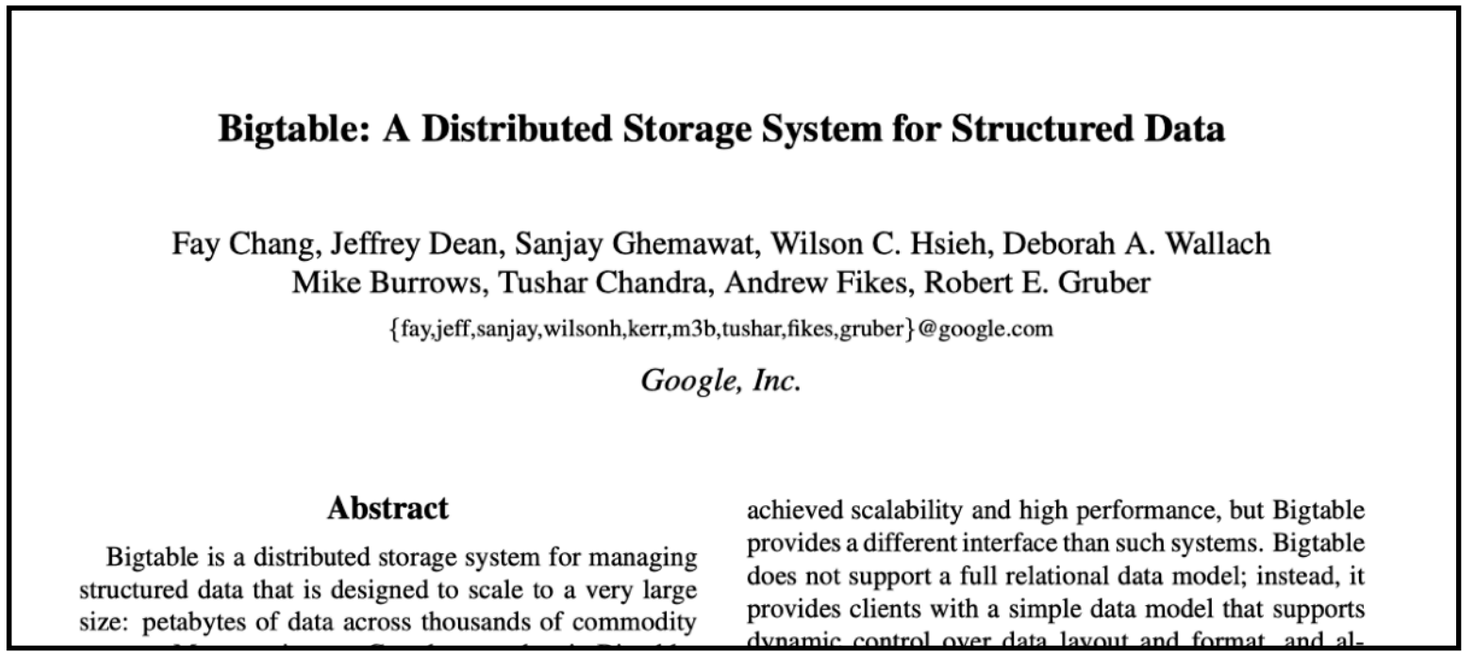
API

**Building blocks**

Implementation

Performance

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# Building blocks

**GFS**: raw storage

**SSTable**: persistent ordered immutable KV map

**Cluster scheduler**: schedules jobs onto machines

- **Borg**: Google's proprietary cluster management system

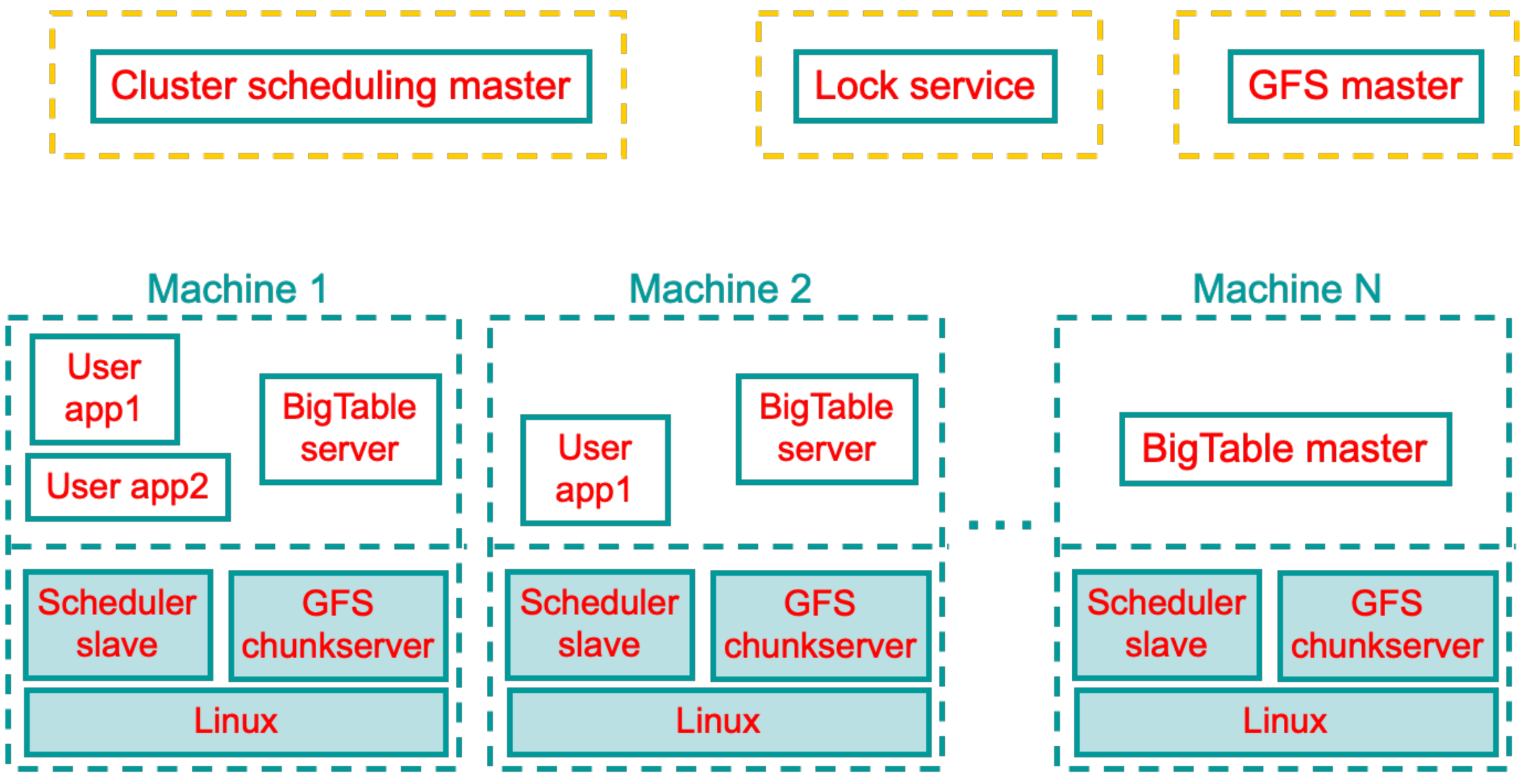
**Lock service**:

- **Chubby**: persistent distributed lock manger, which also can reliably hold tiny files (100s of bytes) w/ high availability

**MapReduce**:

- A distributed big data compute framework often used to read/write BigTable data (covered in the next chapter)

# Typical Cluster



# Outline

Motivation and goals for BigTable

Data model

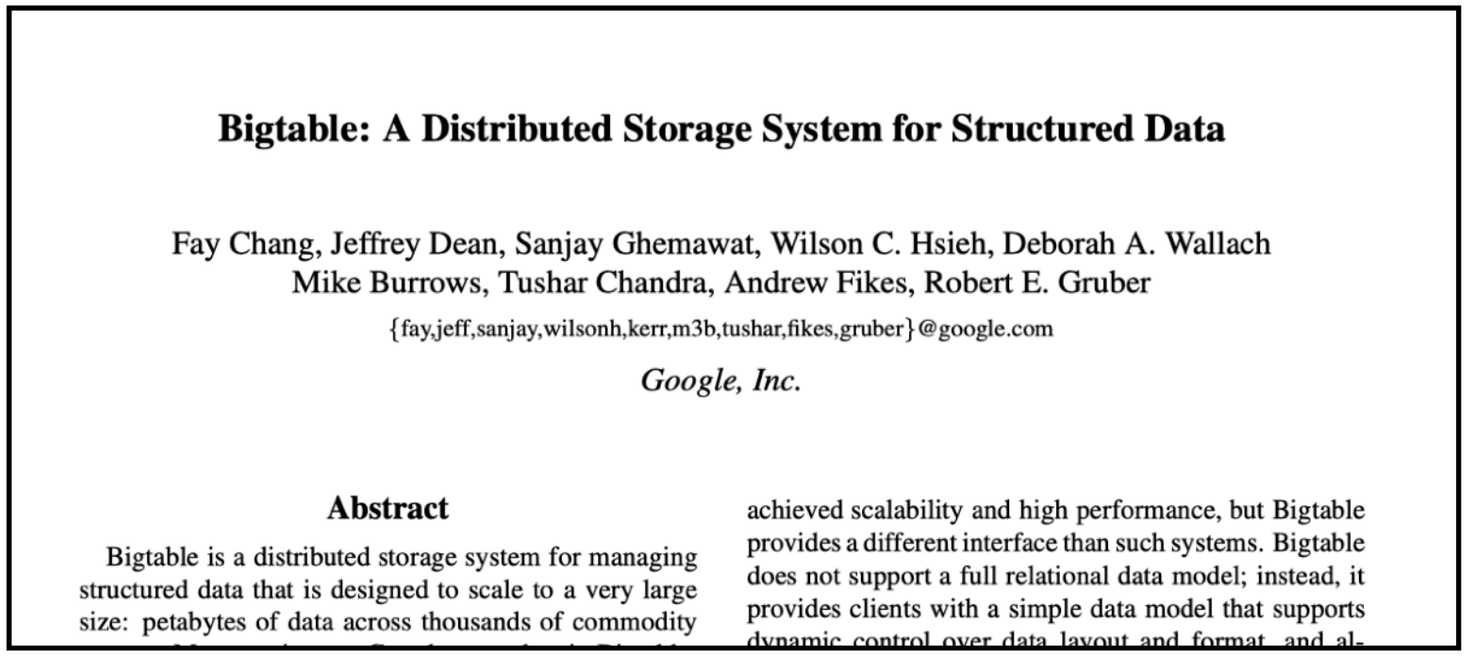
API

Building blocks

**Implementation**

Performance

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# Three Major Components

A library that is linked into every client

A single master server

- Performs DB schema operations
- Dynamically partitions tables across data (tablet) servers
- Handles addition and removal of tablet servers

| | | anchor | contents | language |
|---|---|---|---|---|
| Tablet 1 | ca.mylook | | | |
| Tablet 2 | com.cnn.www | cnnsi.com, $t_4$: CNN<br>cnnsi.com, $t_2$: CNN<br>mylook.ca, $t_1$: CNN.com | $t_6$: <html>…<br>$t_5$: <html>…<br>$t_3$: <html>… | EN |
| | com.cnn.www/ca | | | |
| Tablet 3 | com.cnnsi.com | | | |

- A tablet is a unit of distribution and load balancing
  - Each tablet served by a single tablet server
- Users select keys to control placement of related rows
  - Nearby rows will usually be served by same server

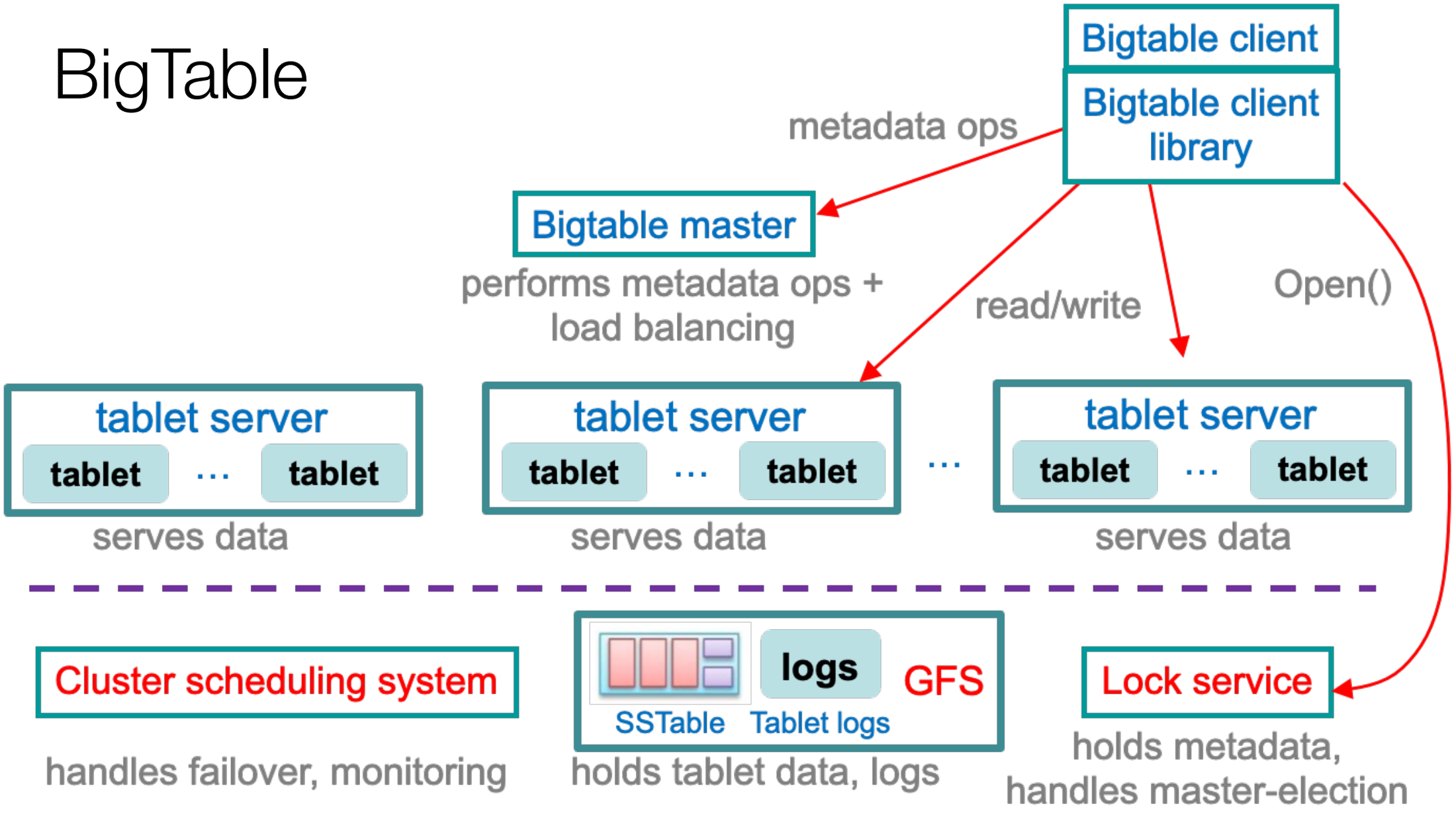
# Three Major Components

Many data (tablet) servers

- Each tablet server serves 10-1000 tablets
- Use GFS for storage and replication
  - colocate w/ GFS chunkservers
- Handle read and writes and splitting of tablets
- Provide low-latency access using **write-optimized** data store
  - use log-structured merge (LSM) tree
- Clients access data from tablet servers directly

# System Architecture

BigTable



# BigTable Storage

Use GFS to store log and data files

- SSTable (Sorted String Table) file format (discussed later)

Use **Chubby** distributed lock service for coordination

- Store bootstrap location of BigTable data
- Store schema metadata (e.g., column families for each table)
- Store access control lists
- Help ensure at most one active master exists
- Help keep track of live tablet servers

# 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

One approach: could use the BigTable master

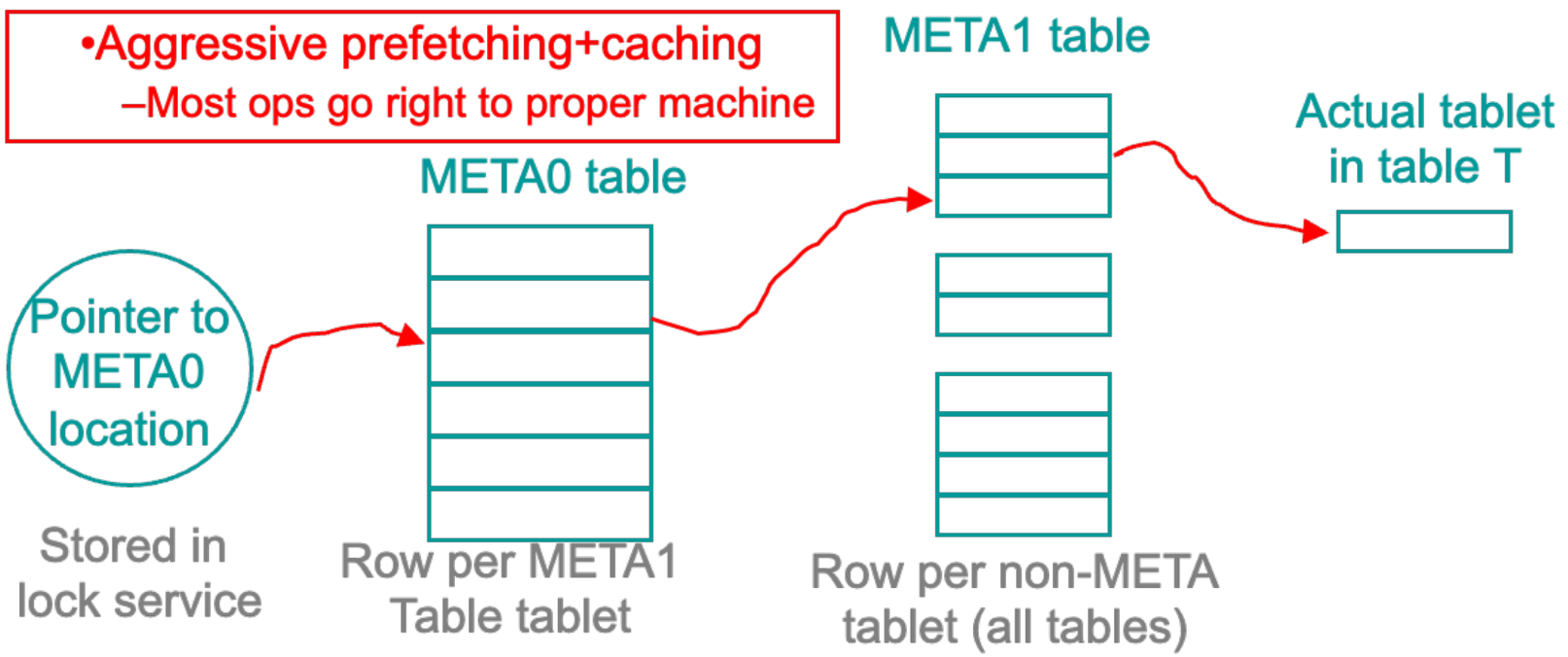
- Central server almost certainly would be bottleneck in large system

Instead: store special tables containing tablet location info in BigTable cell itself

# Locating Tablets

Approach: 3-level hierarchical lookup scheme for tablets

- Location is **ip:port** of relevant server, **rowkey=encoding(tableID+endrow)**
- 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



# Tablet Assignment

Each tablet assigned to one tablet server

BigTable master

- Keeps track of unassigned tablets
- Assigns tablet by tablet load request to a tablet server with sufficient room
- detects when a tablet server no longer is serving its tablets (status of lock in Chubby) and adds tablets to set of unassigned tablets

# Tablet Assignment

Master startup actions

- Acquires a master lock in Chubby (ensures a single master)
- Acquires list of live tablet servers from Chubby
- Gets list of tablets served by asking each tablet server
  - These are assigned tablets
- Scans the master table (METADATA table) to find all tablets
  - unassigned tablets = all tablets - assigned tablets
- Assigns the unassigned tablets to tablet servers

# Tablet Storage Layout

The tablet data and logs are stored in GFS files

- How should the data be stored in the GFS files?

Problem

- GFS supports fast file appends, but not overwrites
- GFS supports large file reads and writes
- However, modern web applications require support for both
  - fast indexed small reads, scans (search rows)
  - high-throughput updates (insert rows)

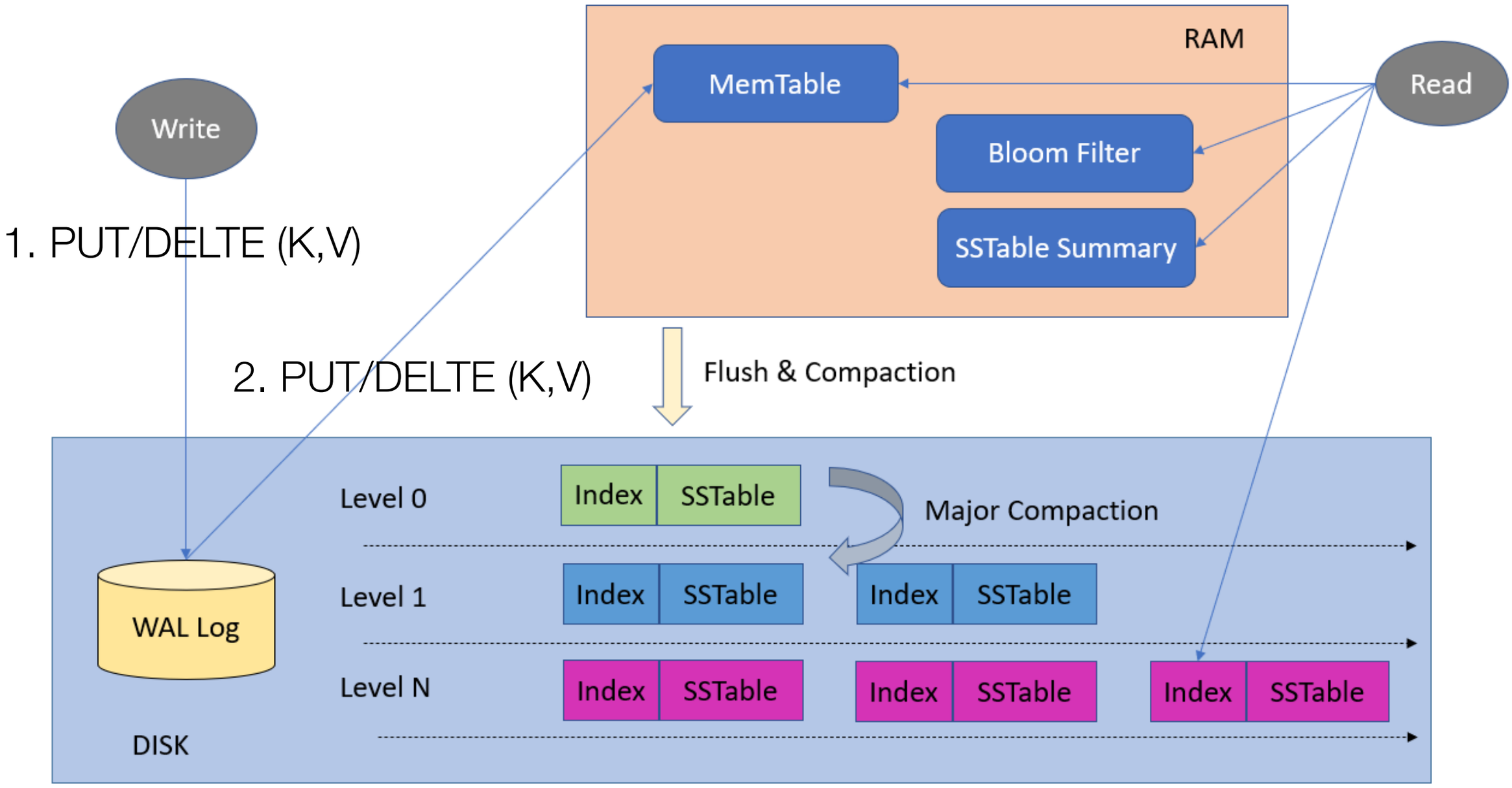
# Log-Structured Storage

Instead of storing KV tuples and updating them in-place, the storage maintains a log that records changes to tuples

- Each log entry represents a tuple **PUT/DELETE** operation
- Originally proposed as log-structured merge trees (LSM-trees) in 1996

Applies changes to an in-memory data structure (***MemTable***) and then writes out the changes sequentially to disk (***SSTable***)

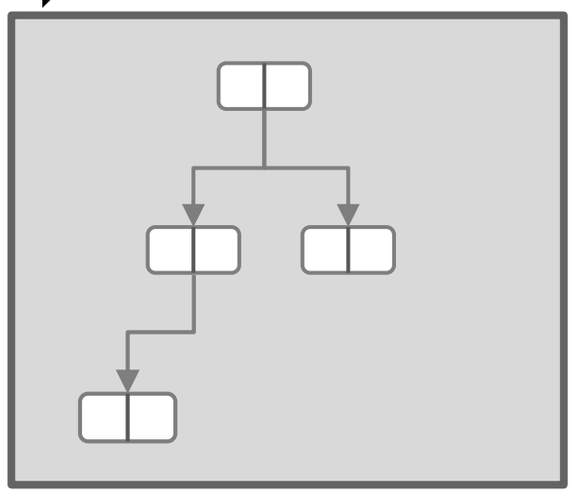
# LSM-Tree Storage




Write-Ahead Log (WAL) for fault tolerance

# LSM-Tree Storage

PUT (key101, $a_1$) ⇨ *MemTable*

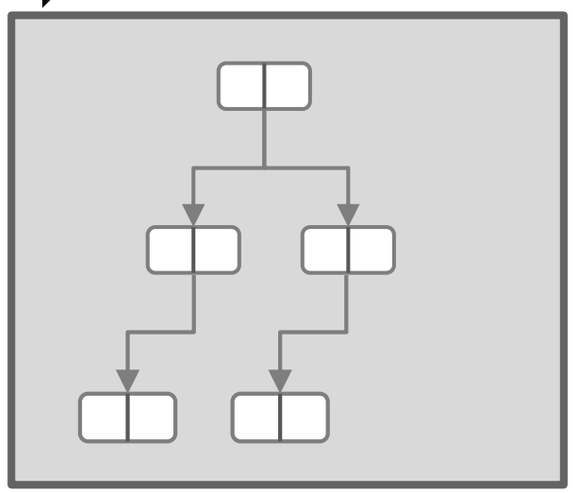


Memory

Disk

# LSM-Tree Storage

PUT (key102, $b_1$) ⇨ *MemTable*

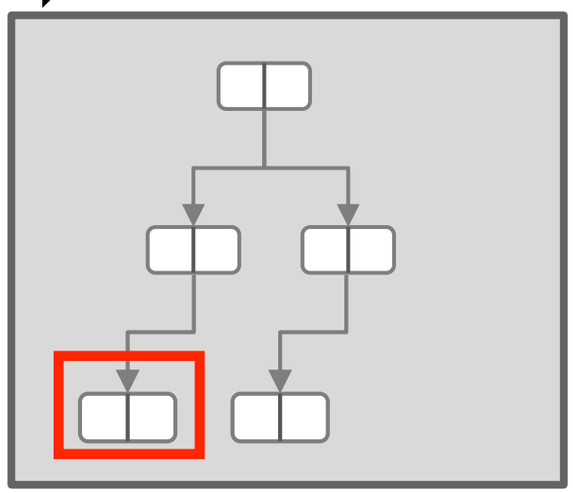


**Memory**

**Disk**

# LSM-Tree Storage

PUT (key101, $a_2$) ⇨ *MemTable*



Memory

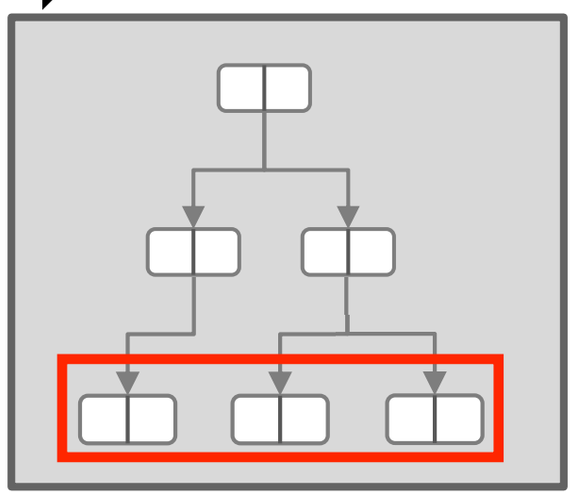
Disk

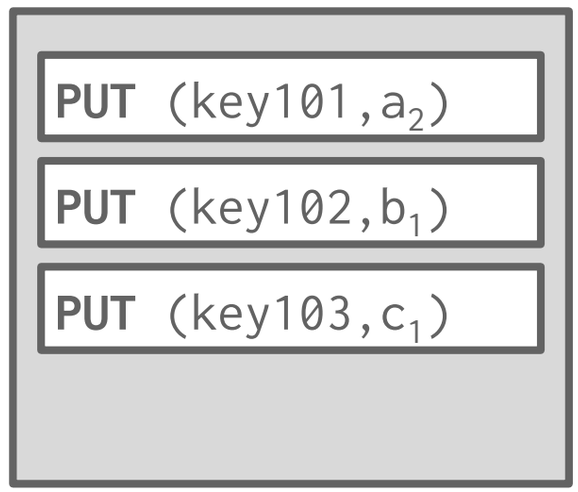
# LSM-Tree Storage

PUT (key103, $c_1$)

Memory

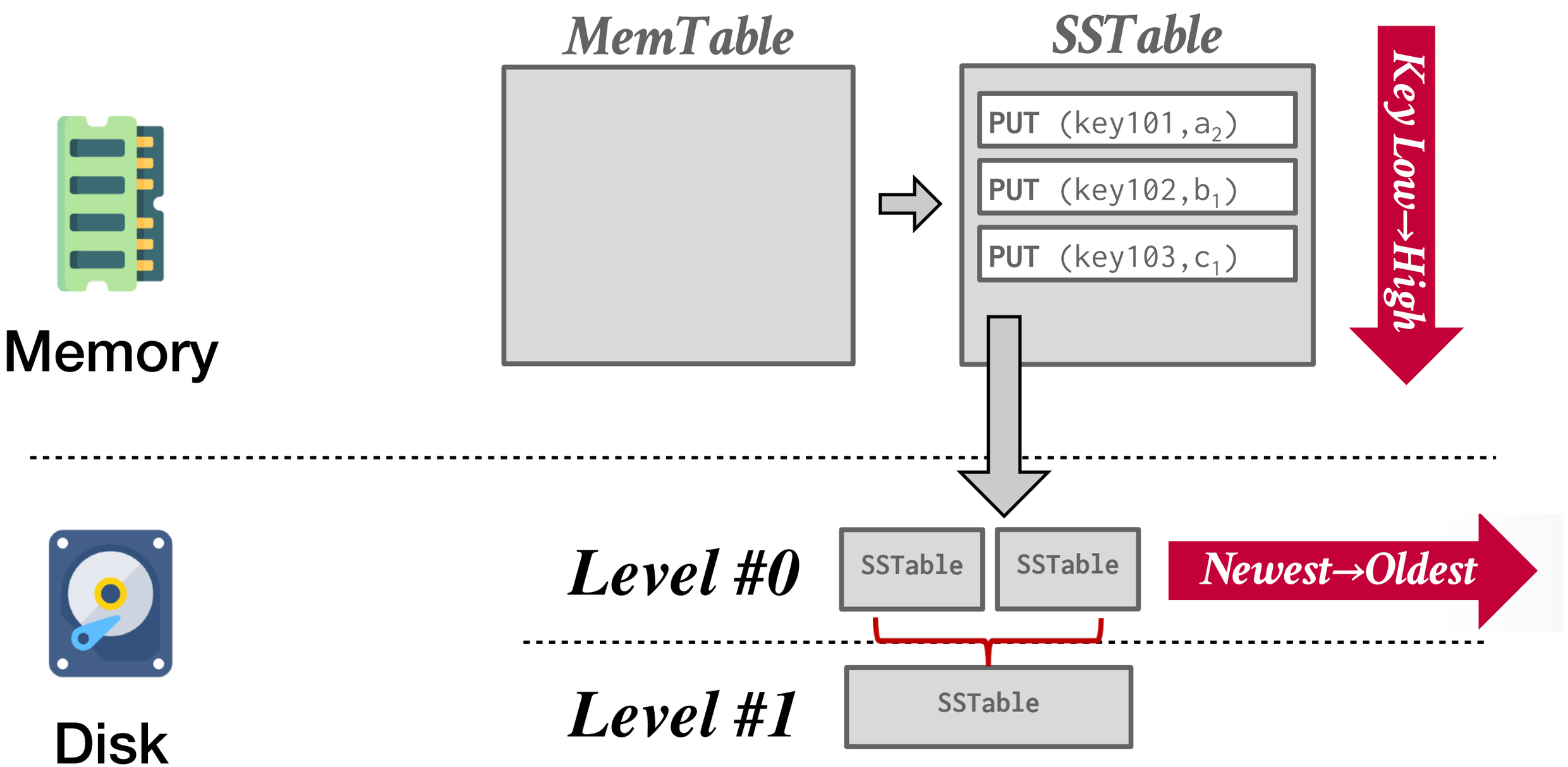
Disk



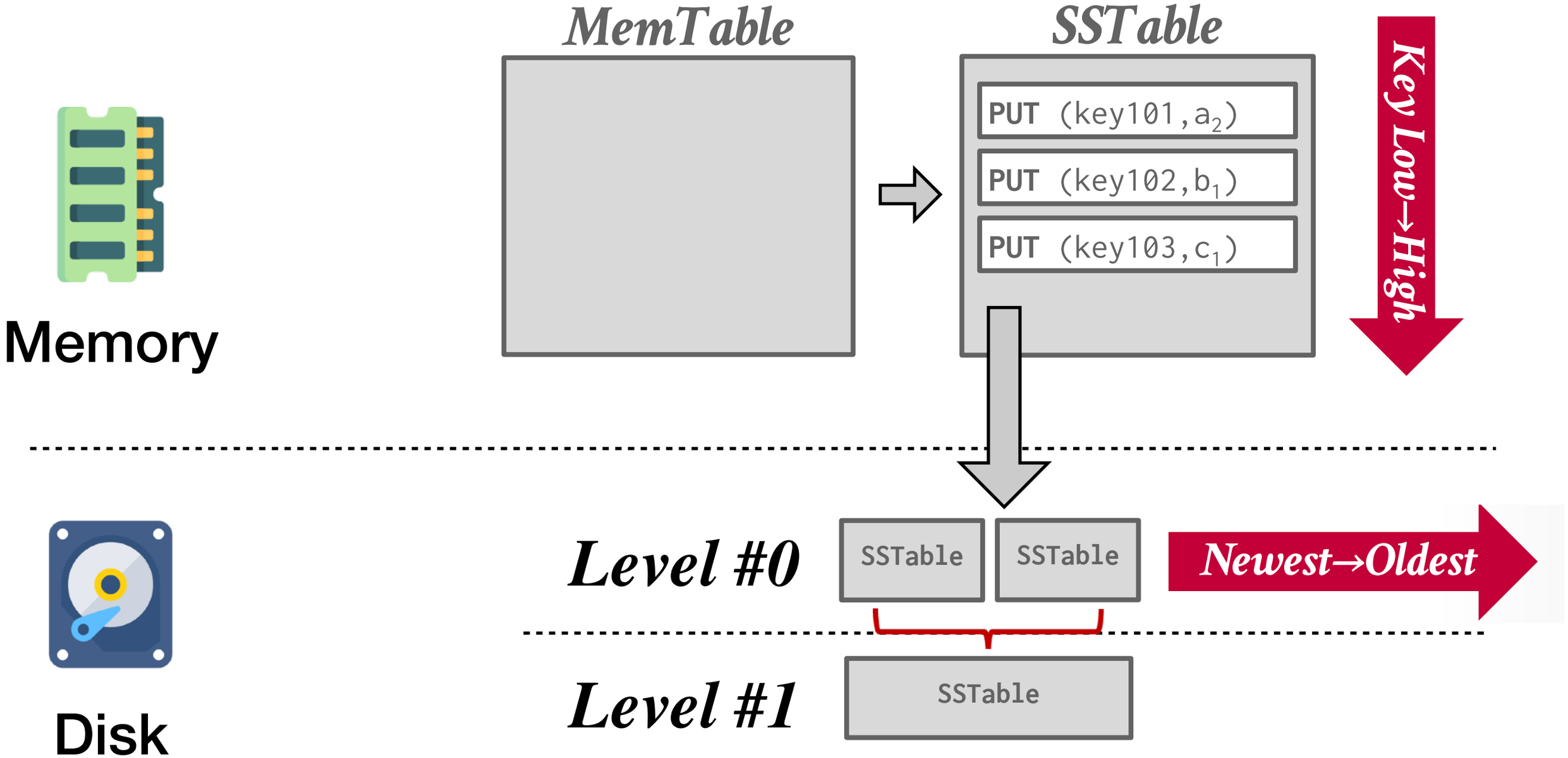




Key Low→High

# LSM-Tree Storage



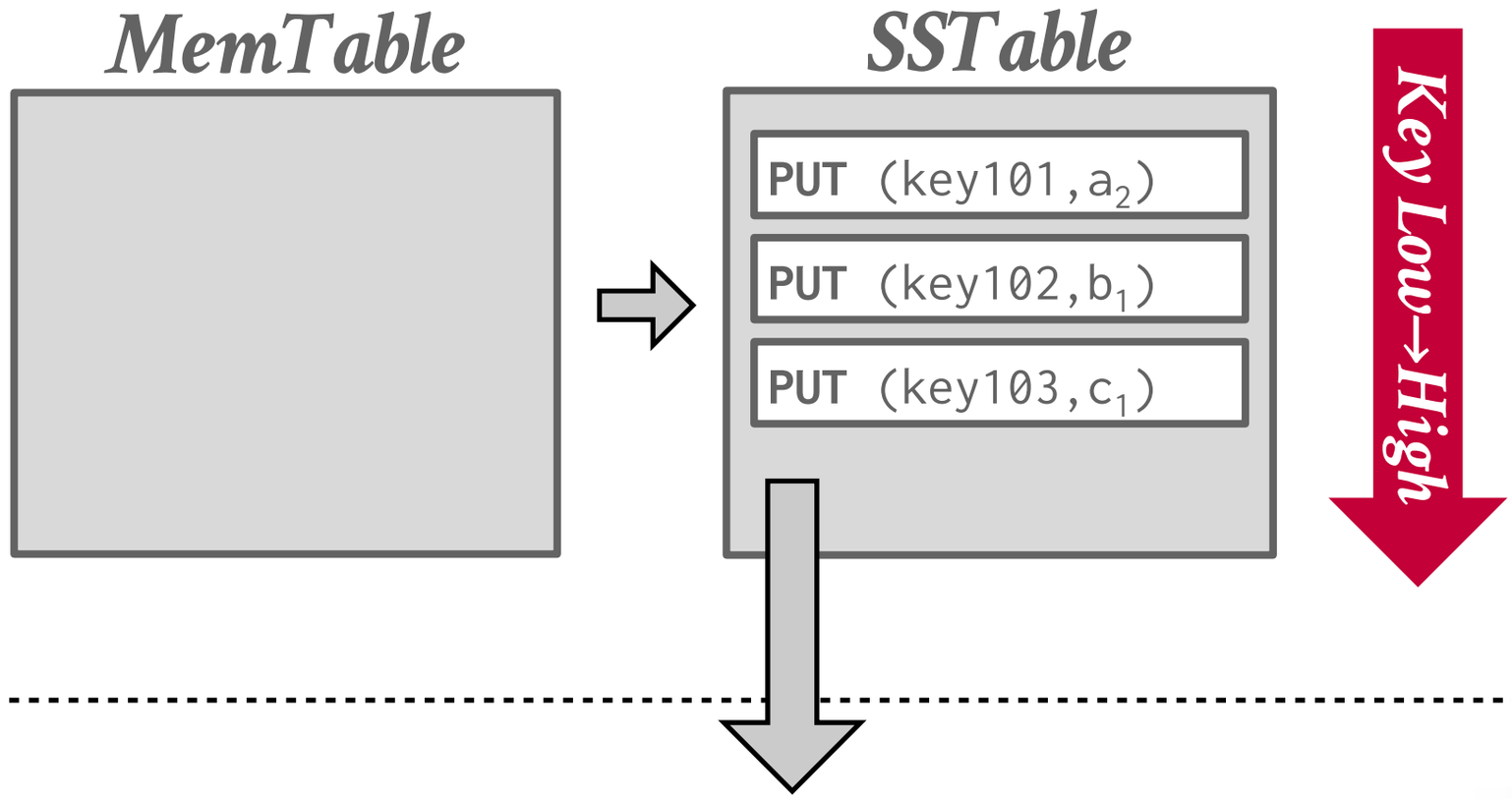
# LSM-Tree Storage



# LSM-Tree Storage

Memory

Disk



# LSM-Tree Storage

Memory

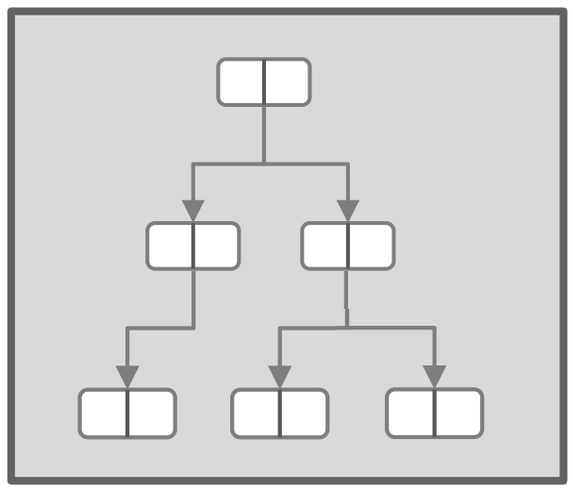
Disk

GET (key101)

*SummaryTable*

- Min/Max Key Per SSTable
- Key Filter Per Level

*MemTable*



*Level #0* SSTable

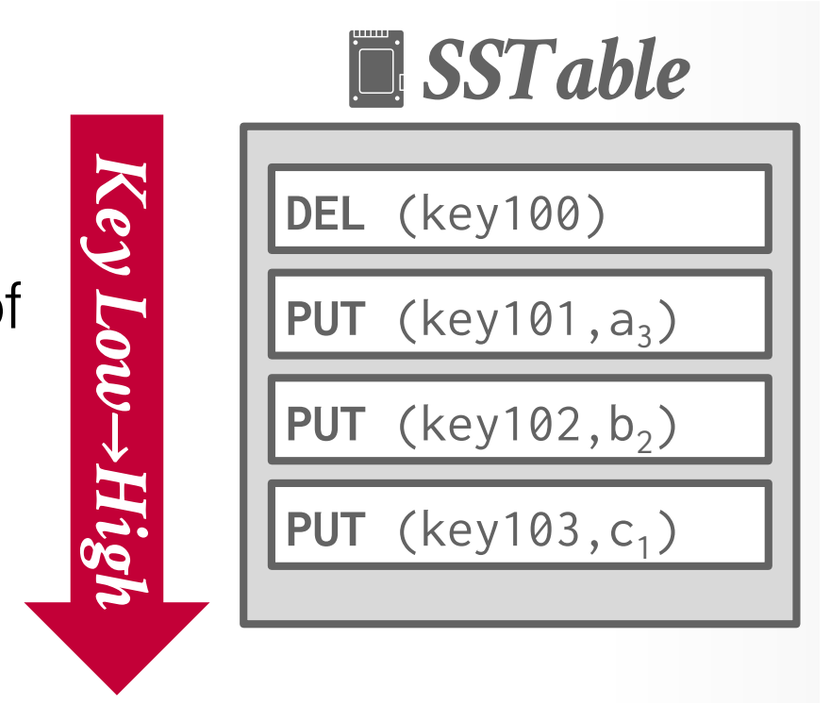
*Level #1* SSTable

*Level #2* SSTable

# LSM-Tree Storage

Uses logging + sorted structure

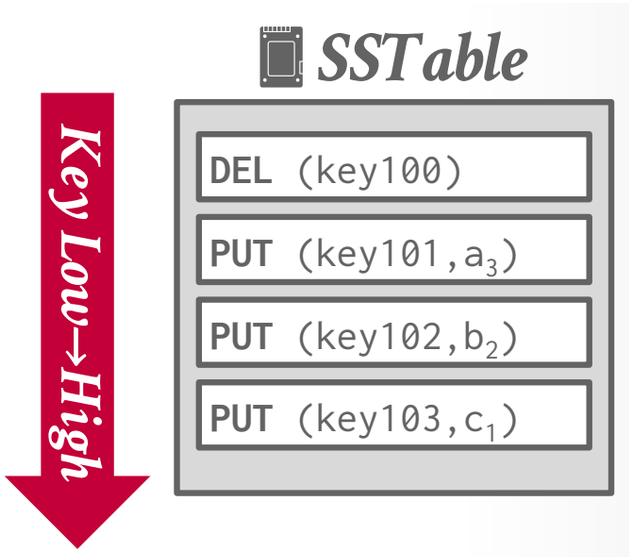
- Only ***MemTable*** allows reads and writes
- All ***SSTables*** are sorted, immutable files of log records (**PUT, DELETE**)
  - Contain versioned (timestamped) data
  - The storage appends log records to the end of the file without checking previous log records
- Allows asynchronous deletes
  - A delete is a new version (tombstone)
  - Previous versions deleted asynchronously during compaction

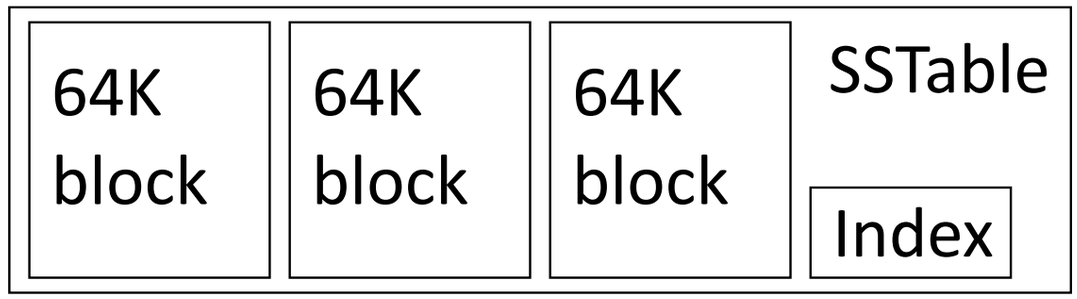


# SSTable

Immutable, sorted file of key-value pairs

- key is (row, column, timestamp)
- Contains blocks of data and an index
  - Index maps key range to block
  - Index loaded into memory when SSTable is opened



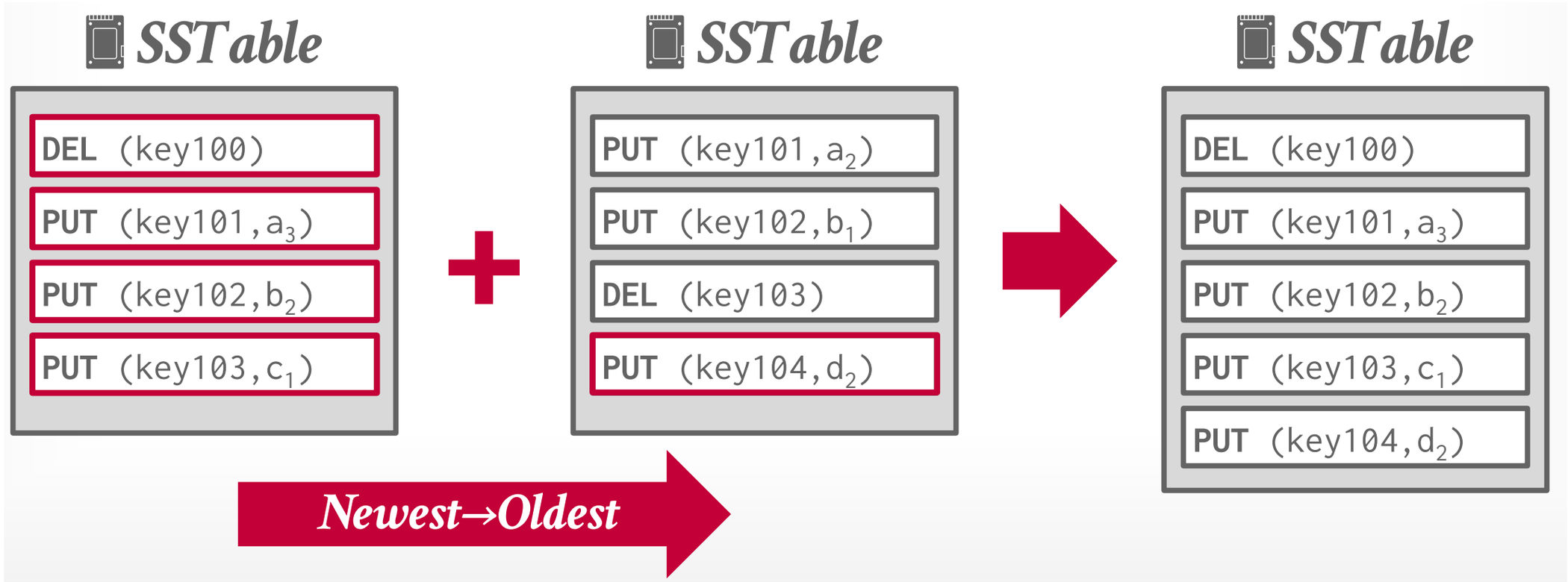




- Key lookup requires a single disk seek, per SSTable
  - Read block into memory (slow)
  - Look up key using binary search within block (fast)

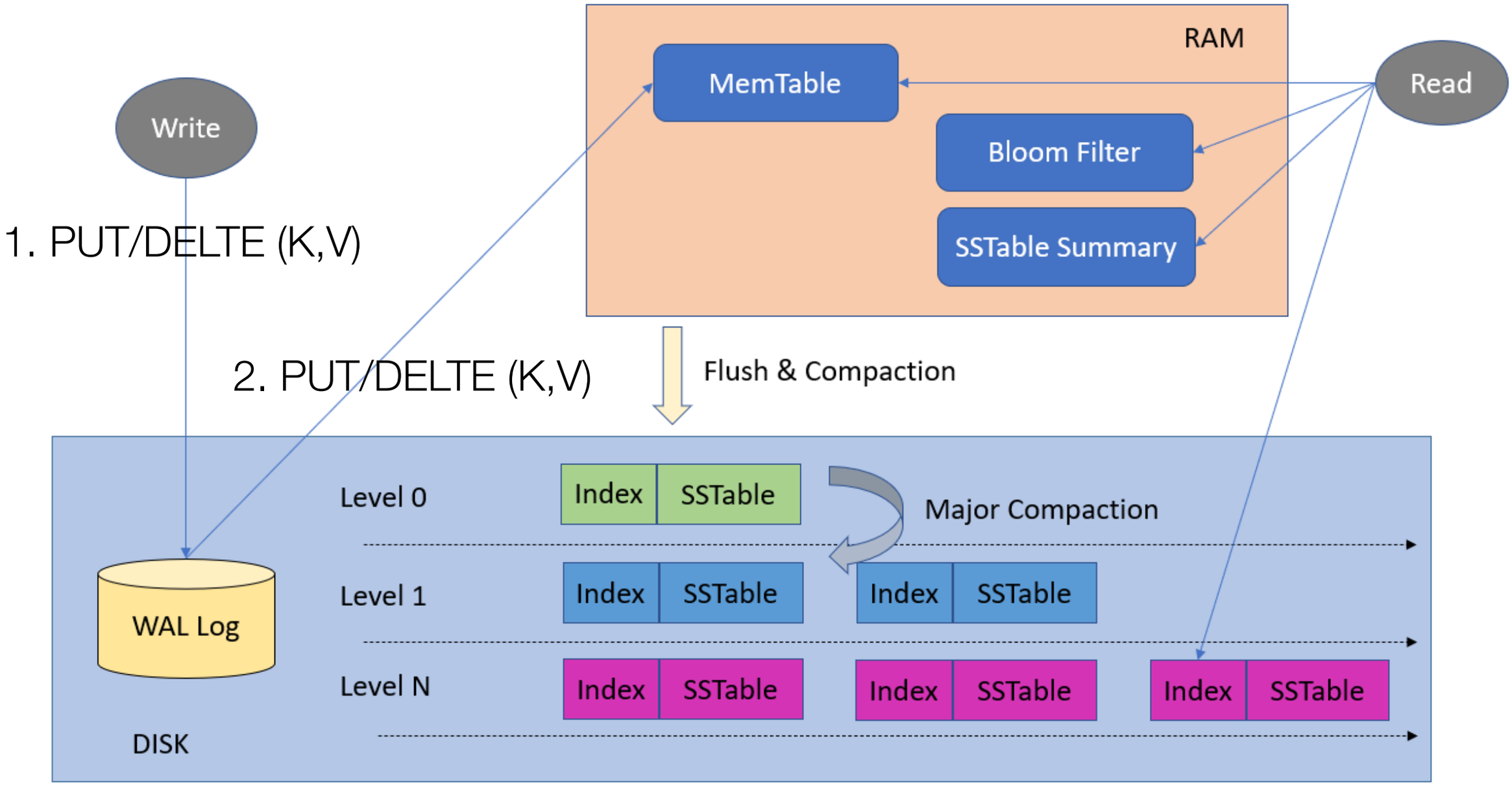
# Compaction

Periodically compact SSTables to reduce wasted space and speed up reads

- Only keep the “latest” values for each key using a sort-merge algorithm



# Putting It Altogether




Write-Ahead Log (WAL) for fault tolerance

# Optimizing Reads: Caching

Cache reads at tablet servers with two-level caching

Scan cache

- Cache key-value pairs from SSTable
- Temporal locality

Block Cache

- SSTable blocks read from GFS
- Spatial locality

# Optimizing Reads: Bloom Filters

Reads need to read from multiple SSTables that make up table

Each SSTable stores a **bloom filter**

- A space-efficient data structure that returns true when the (key, value) pair exists in the SSTable
- But may return false positive (no false negative)

Dramatically reduce disk accesses when the SSTable doesn't have matching (key, value) pair

# Optimizing Writes

Use a single commit log per tablet server, not one per tablet

- Reduces the number of files written, improves seek locality, reduces overhead, etc.
- Different files would mean writes to different locations on disk

Complicates recovery after server fails, since tablets may be loaded on many live tablet servers

- Few log entries associated with any one tablet in the log
- Run a parallel sort by key, then log entries for each tablet are close together

# 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): read small portions without decompressing entire file
- User-specified compression formats

# Outline

Motivation and goals for BigTable

Data model

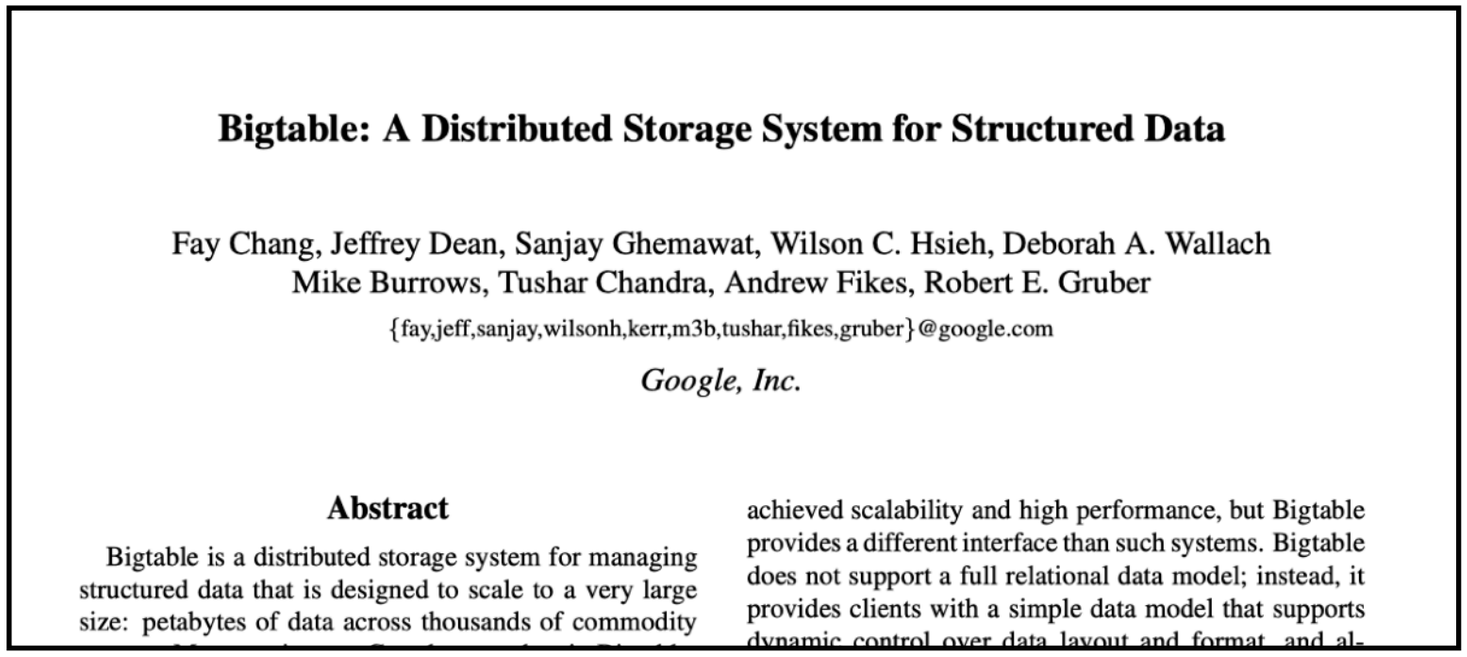
API

Building blocks

Implementation

**Performance**

Conclusion



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

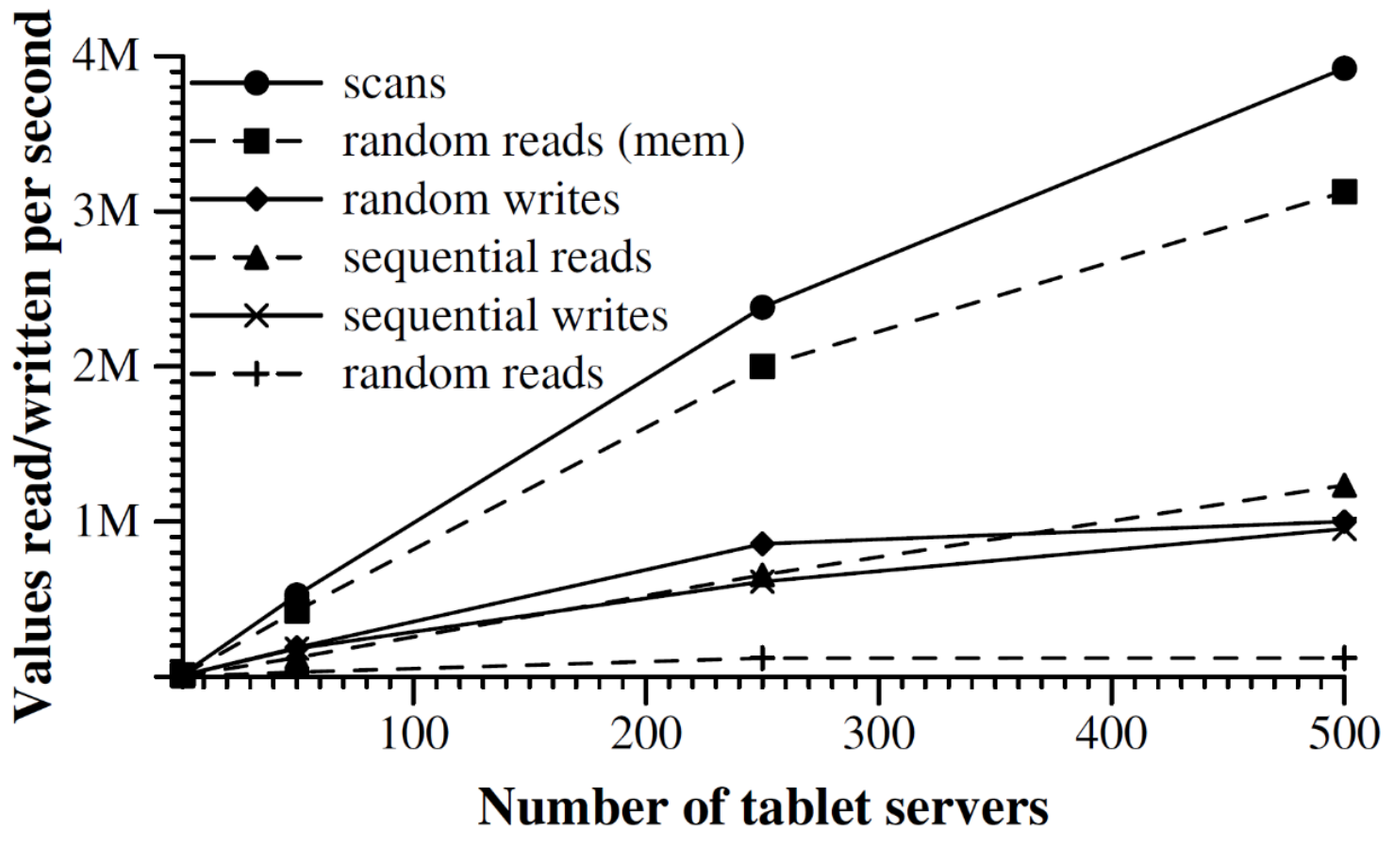
# Performance

Random reads are much slower than all other operations

Sequential reads/writes and random writes are comparable

Random reads from memory are much faster

Scans (sequential reads) are even faster



# Outline

Motivation and goals for BigTable

Data model

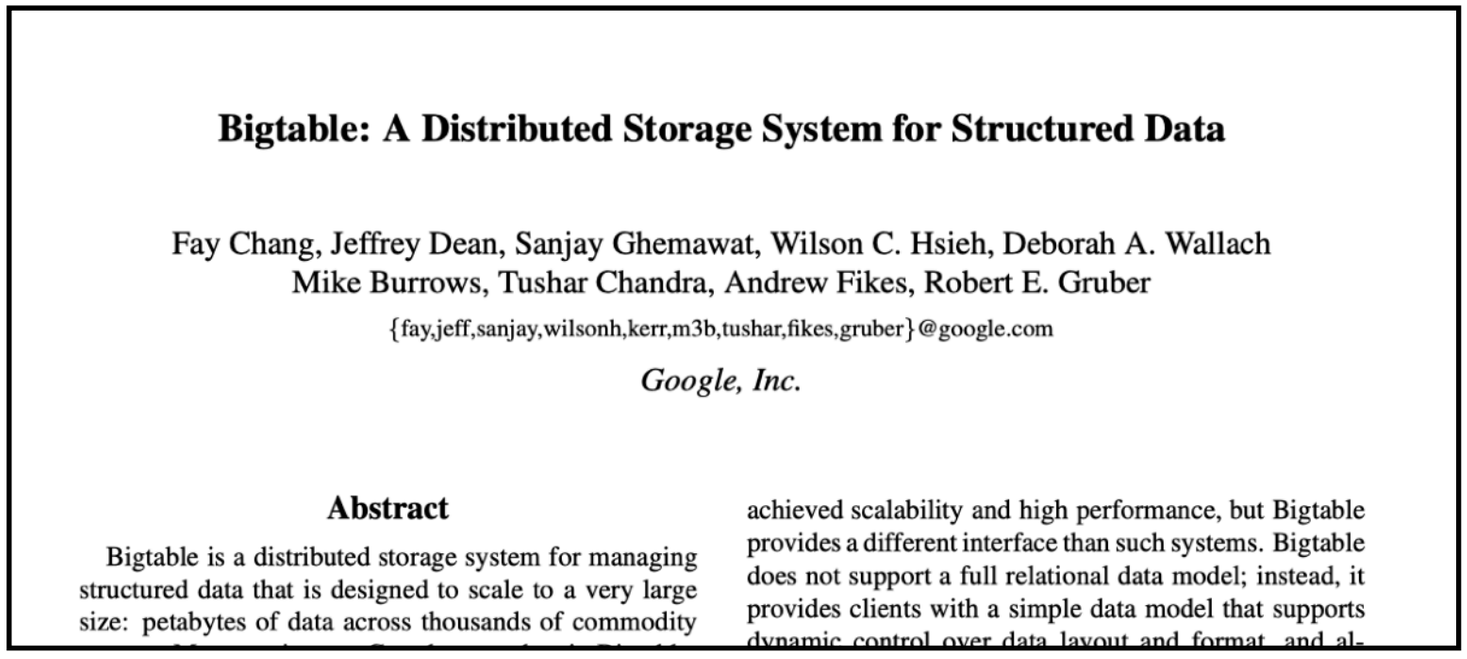
API

Building blocks

Implementation

Performance

**Conclusion**



**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
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

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

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and al-

The very first work on cloud-scale storage for semi-structured data, published in OSDI ’06

# Pros and Cons

Pros

- Can handle massive data and massive objects scalable
- Supports low-latency access for small data sizes
- Supports tables with thousands of columns efficiently
- Allows applications to control data locality

Cons

- Weak consistency model (row-level atomic updates)
  - No table-wide integrity constraints
  - However, sufficient for many applications
- Writing large objects causes much write amplification

# Some Lessons Learned

Many types of failures possible, not only fail-stop

- Memory and network corruption, large clock skew, hung machines, bugs in other systems, extended and asymmetric network partitions, planned and unplanned hardware maintenance
- Big systems need constant system-level monitoring

Delay adding new features until needed

- E.g., initially planned for multi-row transaction APIs

# Conclusion

BigTable is a highly available and scalable cloud database

- Easy to scale by adding tablet servers to the system
- Separating storage from serving data simplifies design, fault tolerance, self management, etc.

If you are Google

- Significant advantages of building own storage system
- Data model applicable to many of its applications

Broad impact

- Apache HBase is an open-source implementation of BigTable
- Apache Cassandra offers BigTable data model

# Credits

Some slides are adapted from Prof. Ashvin Goel's course ECE1724 at the University of Toronto and Prof. Andy Pavlo's course 15-445/645 at CMU