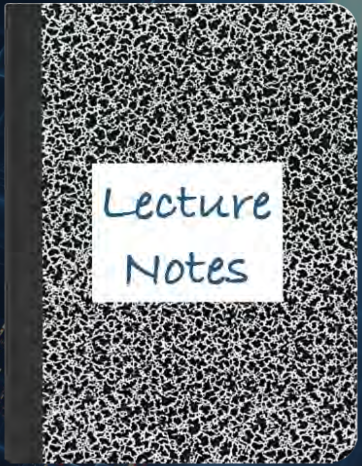
CS 417 – DISTRIBUTED SYSTEMS

Week 9: Distributed Databases

Part 1: Google Bigtable



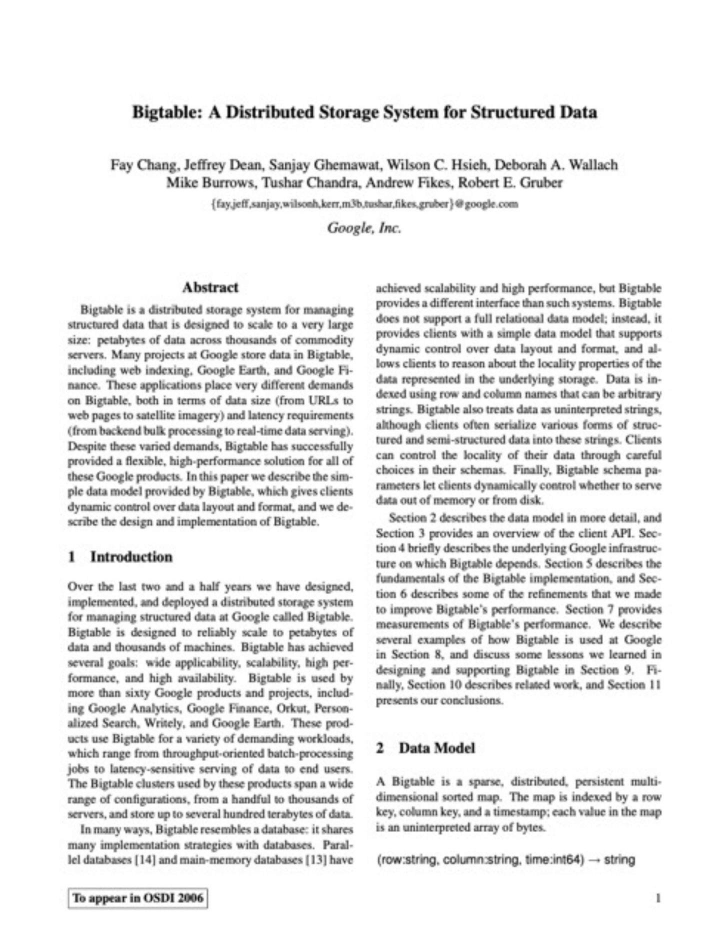
Paul Krzyzanowski

Bigtable

- Highly available distributed storage

- Built with semi-structured data in mind
  - URLs: content, metadata, links, anchors, page rank
  - User data: preferences, account info, recent queries
  - Geography: roads, satellite images, points of interest, annotations

- Large scale
  - Petabytes of data across 100s of thousands of servers
  - Billions of URLs with many versions per page
  - Hundreds of millions of users
  - Thousands of queries per second
  - 100TB+ satellite image data



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 servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have 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 allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

Uses

At Google, used for:

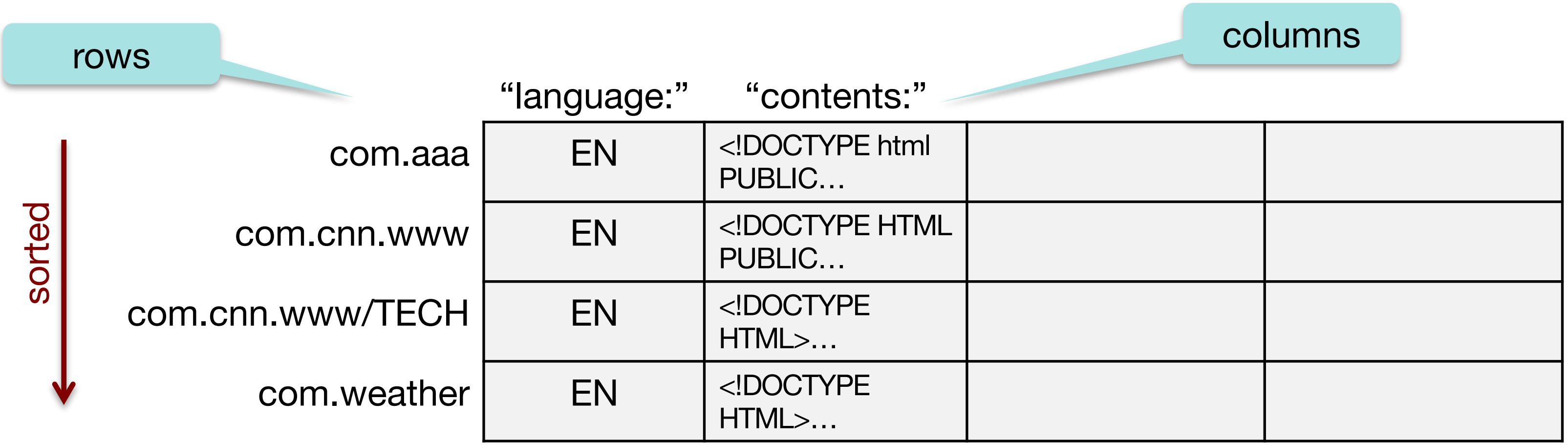
- Google Analytics
- Google Finance
- Personalized search
- Blogger.com
- Google Code hosting
- YouTube
- Gmail
- Google Earth & Google Maps
- Dozens of others… *over sixty products*

A big table

Bigtable is NOT a relational database

Bigtable appears as a large table

“A Bigtable is a sparse, distributed, persistent multidimensional sorted map”*



	“language:”	“contents:”		
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…		
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		
com.weather	EN	<!DOCTYPE HTML>…		

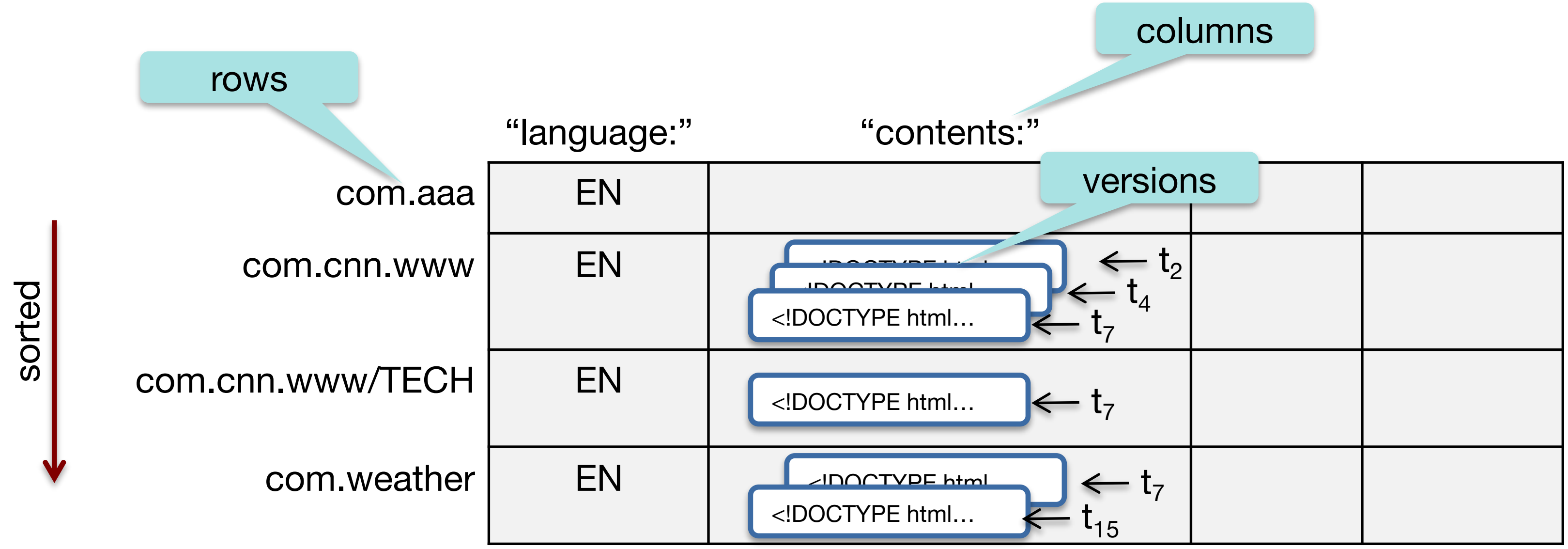
Web table example

*Bigtable: OSDI 2006

Table Model

(row, column, timestamp) → cell contents

- Contents are arbitrary strings (arrays of bytes)



	"language:"	"contents:"		
com.aaa	EN			
com.cnn.www	EN	<!DOCTYPE html… ← t_2 <!DOCTYPE html… ← t_4 <!DOCTYPE html… ← t_7		
com.cnn.www/TECH	EN	<!DOCTYPE html… ← t_7		
com.weather	EN	<!DOCTYPE html… ← t_7 <!DOCTYPE html… ← t_{15}		

Web table example

Columns and Column Families

Column Family = group of related columns ⇒ basic unit of data access

– Data in a column family is typically of the same type

– Implementation of Bigtable compresses data in the same column family

- **Operations**
  - (1) Create column family ⇒ this is an admin task done when the table is created
  - (2) Create a column and store data within the family ⇒ this can be done anytime

- There will typically be a small number of column families
  - ≤ hundreds of column families
  - A table may have an unlimited # of columns within a column family: *often sparsely populated*

- Columns are identified by family:qualifier

Column Families: example

Three column families

- “language:” – language for the web page
- “contents:” – contents of the web page
- “anchor:” – contains text of anchors that reference this page
  - `www.cnn.com` is referenced by Sports Illustrated (`cnnsi.com`) and My-Look (`mlook.ca`)
  - The value of (“`com.cnn.www`”, “`anchor:cnnsi.com`”) is “`CNN`”, the reference text from `cnnsi.com`.

Column family *anchor*

sorted

	“language:”	“contents:”	anchor:cnnsi.com	anchor:mylook.ca
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…	“CNN”	“CNN.com”
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		
com.weather	EN	<!DOCTYPE HTML>…		

Tables & Tablets

- A table is partitioned dynamically by rows into one or more **tablets**

- Tablet = range of contiguous, sorted rows in a table
  - Unit of distribution and load balancing
  - Nearby rows will usually be served by the same server
    - Accessing nearby rows requires communication with a small # of machines
  - You need to choose row keys carefully to ensure good locality
    - E.g., reverse domain names – so the the same domains are adjacent:

      `com.cnn.www` instead of `www.cnn.com`

- Row operations are atomic

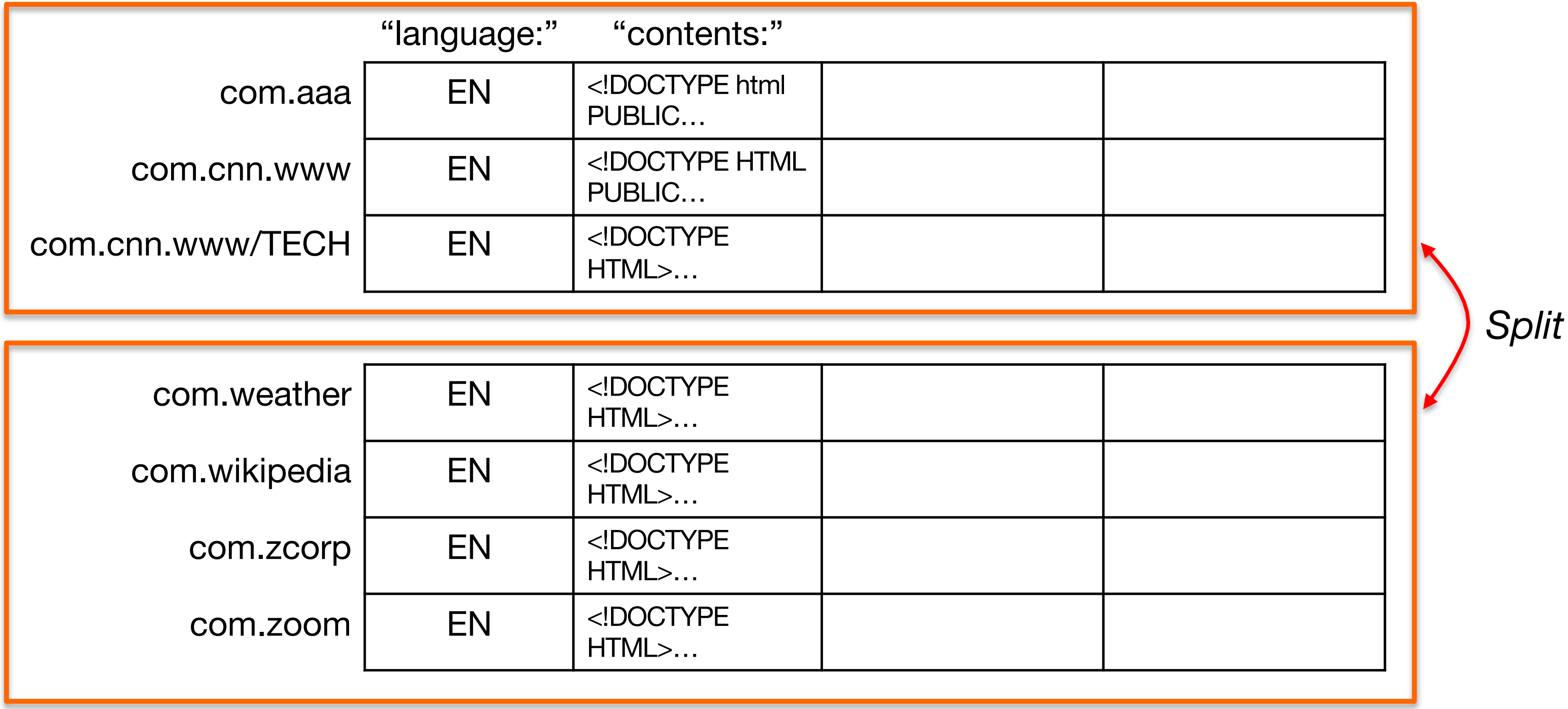
Table splitting

- A table starts as one tablet
- As it grows, it is split into multiple tablets
  - Approximate size: 100-200 MB per tablet by default

	“language:”	“contents:”		
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…		
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		
com.weather	EN	<!DOCTYPE HTML>…		

tablet

Splitting a tablet



	“language:”	“contents:”		
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…		
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		

com.weather	EN	<!DOCTYPE HTML>…		
com.wikipedia	EN	<!DOCTYPE HTML>…		
com.zcorp	EN	<!DOCTYPE HTML>…		
com.zoom	EN	<!DOCTYPE HTML>…		

Timestamped versions

- Each column may contain multiple versions of data

- Version indexed by a 64-bit timestamp
  - Real time or assigned by client

- Per-column-family settings for garbage collection
  - Keep only latest *n* versions
  - Or keep only versions written since time *t*

- Retrieve most recent version if no version specified
  - If specified, return version where timestamp ≤ requested time

API: Operations on Bigtable

- Create/delete tables & column families
- Change cluster, table, and column family metadata (e.g., access control rights)
- Write or delete values in cells
- Read values from specific rows
- Iterate over a subset of data in a table
  - All columns within a column family
  - Multiple column families
    - E.g., regular expressions, such as `anchor:*.cnn.com`
  - Multiple timestamps
  - Adjacent rows
- Atomic read-modify-write row operations

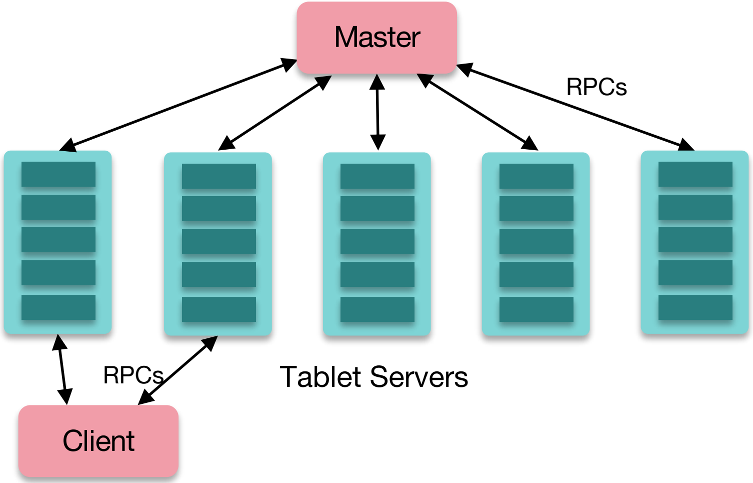
Implementation

One master, many tablet servers

1. **Many tablet servers – coordinate requests to tablets**
   - Can be added or removed dynamically
   - Each manages a set of tablets (typically 10-1,000 tablets/server)
   - Handles read/write requests to tablets
   - Splits tablets when too large

2. **One master server**
   - Assigns tablets to tablet server
   - Balances tablet server load
   - Garbage collection of unneeded SSTable files
   - Schema changes (table & column family creation)

3. **Client library**
   - Client data does not move through the master
   - Clients communicate directly with tablet servers for reads/writes



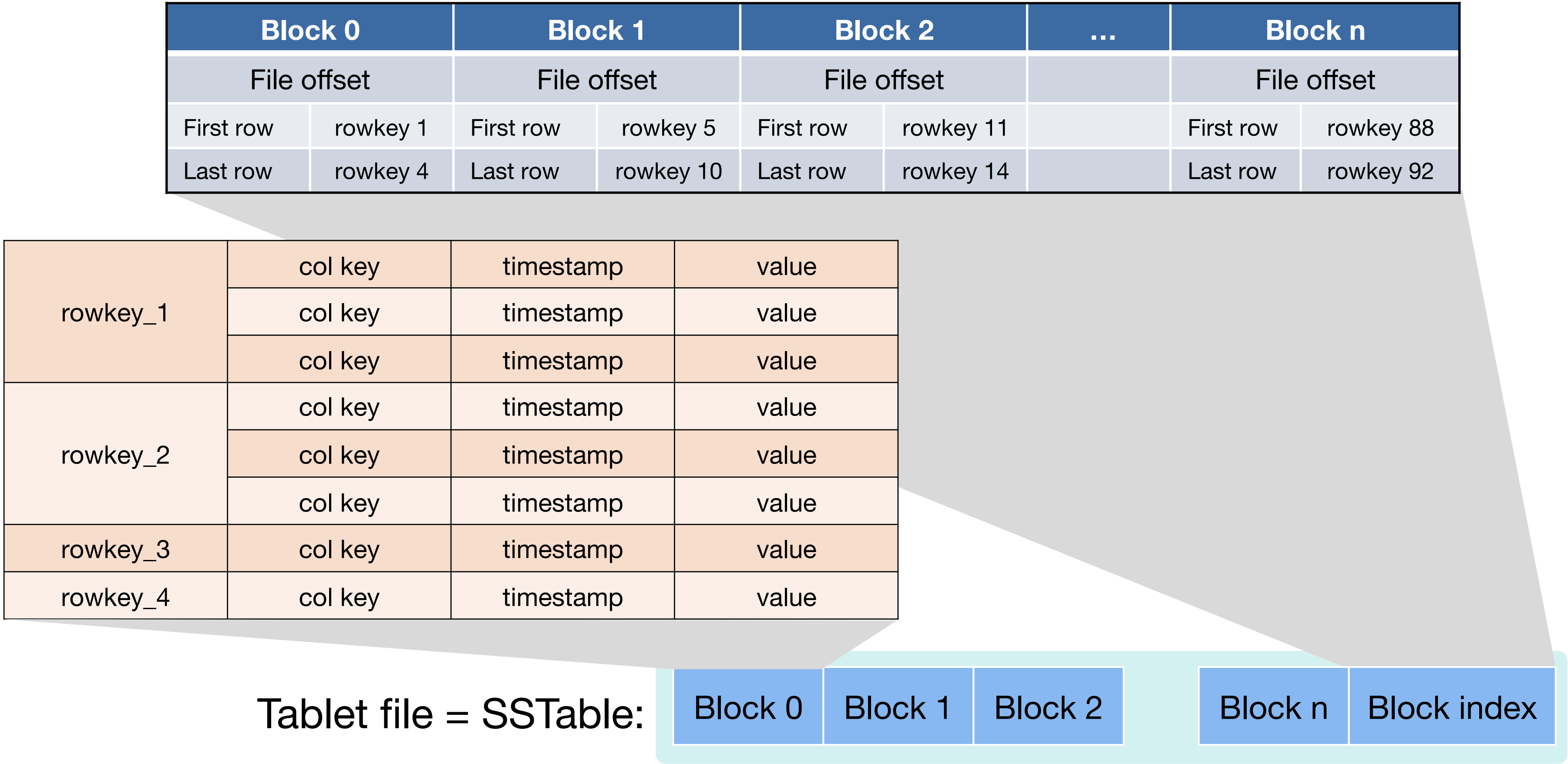
Implementation: Tablets Stored in SSTable

Google SSTable (Sorted String Table)

- Internal file format optimized for streaming I/O and storing <key,value> data
- **Sequence of 64 KB blocks – each block is sorted by rows**
  - Each row contains a list of {column key, timestamp, value} entries
- Index at end of the file and loaded into memory when the table is opened
- Memory or disk-based; indexes are cached in memory
- Provides a persistent, ordered, *immutable* map from keys to values
- Append-only structure
  - If there are additions/deletions/changes to rows
  - New SSTables are written out with the deleted data removed
  - Periodic compaction merges SSTables and removes old retired ones

For a description of SSTable please see https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

Implementation: Tablets Stored in SSTable



Block 0		Block 1		Block 2		...	Block n	
File offset		File offset		File offset			File offset	
First row	rowkey 1	First row	rowkey 5	First row	rowkey 11		First row	rowkey 88
Last row	rowkey 4	Last row	rowkey 10	Last row	rowkey 14		Last row	rowkey 92

rowkey_1	col key	timestamp	value
	col key	timestamp	value
	col key	timestamp	value
rowkey_2	col key	timestamp	value
	col key	timestamp	value
	col key	timestamp	value
rowkey_3	col key	timestamp	value
rowkey_4	col key	timestamp	value

Tablet file = SSTable:

Implementation: Supporting Services

Chubby

- Ensure there is only one active master
- Store bootstrap location of Bigtable data
- Discover tablet servers
- Store Bigtable schema information
- Store access control lists

Cluster management system

- For scheduling jobs, monitoring health, dealing with failures

Implementation: Supporting Services

GFS

- Stores all the tablet files

Chubby is used to:

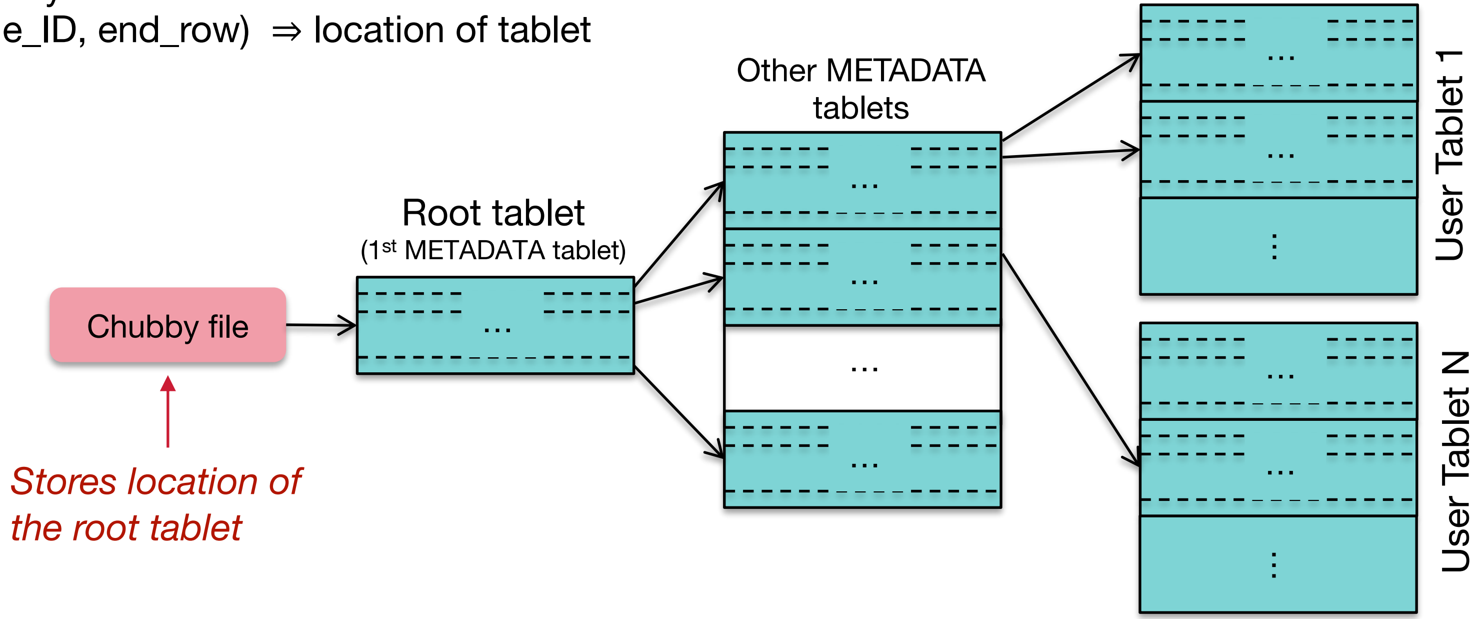
- Enforce single master
- Store bootstrap info
- Discover tablet servers
- Store Bigtable schema
- Store ACLs

- **Cluster management system**
  - For scheduling jobs, monitoring health, dealing with failures

Implementation: METADATA table

Three-level hierarchy

- Balanced B+ tree
- Root tablet contains location of all tablets in a special METADATA table
- Row key of METADATA table contains the location of each tablet
  f(table_ID, end_row) ⇒ location of tablet



Stores location of the root tablet

Startup: server discovery & allocation

When a tablet server starts:

- Creates a unique file name in a Chubby "`servers`" directory

When master starts:

- Grabs a **unique master lock** in Chubby
- Scans the **servers** directory to find live tablet servers
- Contacts **each tablet server** to discover *tablet→server* mapping
- Scans the METADATA table to learn the full set of tablets

Fault Tolerance

Fault tolerance is provided by GFS & Chubby

- Dead tablet server
  - Master is responsible for detecting when a tablet server is not working
    - Asks tablet server for status of its lock
    - If the tablet server cannot be reached or has lost its lock
      - Master attempts to grab that server's lock
      - If it succeeds, then the tablet server is dead or cannot reach Chubby
      - Master moves tablets that were assigned to that server into an unassigned state
- Dead master
  - Master kills itself when its Chubby lease expires
  - Cluster management system detects a non-responding master
- Chubby: designed for fault tolerance (5-way state machine replication)
- GFS: stores underlying data – designed for *n*-way replication

Bigtable Replication

- Each table can be configured for replication to multiple Bigtable clusters in different data centers

- Bigtable uses an *eventual consistency* model for replication
  - Replicas may be updated asynchronously

Sample applications

Google Analytics

- Raw Click Table (~200 TB)
  - Row for each end-user session
  - Row name: {website name and time of session}
    - Sessions that visit the same web site are sorted & contiguous
- Summary Table (~20 TB)
  - Contains various summaries for each crawled website
  - Generated from the Raw Click table via periodic MapReduce jobs

Sample applications

Personalized Search

- One Bigtable row per user (unique user ID)
- Column family per type of action
  - E.g., column family for web queries (your entire search history!)
- Bigtable timestamp for each element identifies when the event occurred
- Uses MapReduce over Bigtable to personalize live search results

Sample applications

- Google Maps / Google Earth
  - Preprocessing
    - Table for raw imagery (~70 TB)
    - Each row corresponds to a single geographic segment
    - Rows are named to ensure that adjacent segments are near each other
    - Column family: keep track of sources of data per segment
      (this is a large # of columns – one for each raw data image – but sparse)
  - MapReduce used to preprocess data
  - Serving
    - Table to index data stored in GFS
    - Small (~500 GB) but serves tens of thousands of queries with low latency

Bigtable outside of Google

Apache HBase

- Built on the Bigtable design
- Small differences (may disappear)
  - Access control not enforced per column family
  - Millisecond vs. microsecond timestamps
  - No client script execution to process stored data
  - Built to use HDFS or any other file system
  - No support for memory mapped tablets
  - Improved fault tolerance with multiple masters on standby

Bigtable vs. Amazon Dynamo

- Dynamo targets apps that only need key/value access with a primary focus on high availability
  - Dynamo: key-value store versus Bigtable’s column-store (column families and columns within them for each key that's accessed)
  - Bigtable: distributed DB built on GFS
  - Dynamo: distributed hash table
  - Bigtable supports iterating over rows in a table
  - Dynamo updates are not rejected even during network partitions or server failures

The End