COMP5338 – Advanced Data Models

Week 11: LSM and Google Bigtable

Dr. Ying Zhou
School of Computer Science

Outline

- **Log Structured Merge Tree**
- **Bigtable Data model**
- **Bigtable Architecture**

DB Queries

- Reading and writing data are the most fundamental functions a database should provide
- Different application domains have different requirements on read and write queries
  - The ratio of read and write queries
    - Read heavy/write heavy/balanced
  - The ratio of different types of read and write
    - Write query: ratio of insert/update/delete query
    - Read query: ratio of random point query and range query
- Different queries have different memory and disk access patterns
- There is no implementation/technique that can simultaneously optimize the performance of all types of queries

DB Query and Performance

- Read vs Write Query
  - E.g Index speeds up read query but slows down write query, so we only build index selectively
- Different Read Queries
  - For point query, hash index provides better performance than tree based index
  - For range query, hash index is useless
- Early database systems try to provide a general solution to relatively balanced query workload
- There are many new systems specialized for a particular workload type

Write Heavy Workload

- Many systems do not maintain traditional business data and do not need transactional processing
  - Business data domains: bank transaction, airline reservation, course enrolment, library record, etc
  - Other domains: system monitoring data, scientific data collected by sensors, user activity data collected by system, etc
- Many application domains require support for large data size, very high write throughput and relatively simple read requests
  - Data are collected and inserted by some application in contrast to initiated by end user transactions
    - Mostly append(insert) type of write
  - Data are analyzed in batch to discover patterns or to make predictions
    - Mostly sequential read

Google Search Engine Data

- A whole copy of the web collected using crawler and done periodically
- Typical features of the data
  - Large data size
  - Frequently inserted into the system
  - Rarely deleted
  - Scanned to build inverted index (word -> page)
  - Page meta data such as links between pages need to be used frequently to compute PageRank score as part of the ranking indicator
- Solution:
  - Build **Bigtable** cluster to store web data

Facebook System Monitoring data

- System measuring data points like *CPU load*, *error rate*, *latency*, etc generated by "thousands of individual systems running on many thousands of machines, often across multiple geo-replicated data centers"
- Run real time queries to identify and diagnose problems as they arise.
- Very high write throughput
- Relatively simple read query that usually scan a range of recent data points
- Solution:
  - Previous: A time Series Database(TSDB) build on top of HBase (the open source version of **Bigtable**)
  - Now: An in-memory TSDB called Gorilla for recent data (26 hours)

Log Structured Merge Tree: Motivation

- Many disk based database systems designed for such write heavy workload (most of them are TSDB) use storage engine based on LSM tree
- LSM was initially proposed in 1996 by Patrick O'Neil et.al.
- The motivation is to provide efficient query on logs of long running transactions
  - Logs are append-only files in <u>time</u> order
  - Querying logs for early transaction events on certain <u>attribute</u> is not efficient
  - There is a need for indexed logs
  - Tree structure (B-tree structure) is the most common index structure

LSM Tree: general solution

- Design principles
  - Memory access is much faster than disk access
  - Memory space is much smaller than disk space
  - At disk level, appending to a file is faster than randomly updating a file
- General solution path
  - Maintain several levels of indexed data (tree) at different storage levels
  - Periodically merge and sort data at different levels
  - Files are only appended (similar to log writing)
- A typical two-level solution
  - Maintain latest entries in memory (C_0), organized as tree structure, for easy query
  - When memory threshold is reached, migrate the in memory data structure to disk as a new file, maintaining the tree structure in the file (C_1)
    - The file is indexed
  - Periodically sort merge the files to create large files while still maintaining the indexed structure (C_1)

LSM Tree: standard implementation

- The 1996 paper gave general solution path but not textbook implementation
- Many terminologies later associated with LSM are proposed in Google's Bigtable paper
  - Memtable, SSTable, Compaction, etc
- Bigtable paper also proposed a detailed enough implementation that are used in many other systems
  - HBase, Cassandra, LevelDB, RockDB, InfluxDB, etc.
  - Most of the systems support key based query and may operate in pure key-value model, wide column model or time series model.
  - Tree structure in memory and disk is just one type of index structure; Bigtable use sorted map instead.

Outline

- Log Structured Merge Tree
- **Bigtable Data model**
- Bigtable Architecture

Data Model

- Bigtable is generally classified as <u>wide-column</u> data model
- "A Bigtable is a **<u>sparse</u>**, distributed, persistent **<u>multidimensional sorted map</u>**"
- Basic concepts: table, row, **column family**, column, timestamp
  - (rowkey:string, columnKey:string, timestampe:int64) **->** value: string
- Example Bigtable to store web pages
  - Stores the data about home page of *cnn* website
    - The <u>URL</u> is "www.cnn.com"
    - The <u>language</u> is "EN"
    - The <u>content</u> is "<html> …</html>"
    - It is <u>referenced</u> by two other pages
      - Sports Illustrated (**cnnsi.com**) , using an anchor text "CNN"
      - My-Look (**my.look.ca**), using an anchor text "CNN.com"

Relational Data Model vs Bigtable Model

web table

url	language	content
“www.cnn.com”	“EN”	“<html> … </html>”

link table

url	referencingUrl	anchorText
“www.cnn.com”	“connsi.com”	“CNN”
“www.cnn.com”	“my.look.ca”	“CNN.com”

column family

column family

column family with two column keys

Row key

	language	content	anchor	
			“connsi.com”	“my.look.ca”
“com.cnn.www”	“EN”	“<html> …”	“CNN”	“CNN.com”

Rows

sorted

Using reversed URL to ensure similar web page would be put in neighboring rows

	language	content	anchor
“com.cnn.www”			
“com.cnn.www/WORLD”			
“com.cnn.weather”			
“com.cts.www”			

- Row keys are arbitrary strings
- Row keys are sorted in lexicographic order
- Large table is dynamically partitioned by row key <u>ranges</u>
  - Row key is partition (sharding) key
  - Each partition is called a **tablet**, and is served by a server
  - Nearby rows will usually be served by the same server
  - Accessing nearby rows requires communication with a small number of machines

Table Splitting

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

	language	content	anchor
“com.cnn.www”			
“com.cnn.www/WORLD”			
“com.cnn.weather”			
“com.cts.www”			

One tablet

Table Splitting (cont'd)

sorted

"com.cnn.www"
"com.cnn.www/WORLD"
"com.cnn.weather"
"com.cts.www" — last key

language	content	anchor

"com.nytimes.www"
"com.seattletimes.www"
"com.washingtonpost.www"
"com.zdnet.www" — last key

language	content	anchor

Columns and Column Families

- Relational model only has “row” and “column” concepts
- Bigtable has “row”, “column” and “column family” concepts
- Column family
  - Just a group of columns with a **printable name**
  - Each **column** inside a **column family** has a **column key**
    - Column key is named as **family:qualifier**
- Column family can be viewed is a convenient way to store “collection” type data at design level

- Data stored in a column family is usually of the same type

Columns and Column Families (cont'd)

- Column Family is part of the **schema definition**
  - When we create a table, we also create a few column families by specifying their names
  - The number of column families in a table is typically small and relatively stable
    - Less than hundred
  - A column family theoretically can have unlimited number of columns
    - The row could be very "wide" with many columns
      - Wide-column store
    - E.g. a popular web page in the web table may be referenced by thousands, or even millions of other pages
    - *Implications*: we may have some tablet storing only one row!

Column Family Examples

- The web table example has three column families
  - “language” -- with only one column to store a web page’s language
    - Each web page can only have one language
    - Just like a normal column in relational table
    - Column key is “**language:**”
  - “content” -- again with only one column to store the actual HTML text
    - Column key is “**content:”**
  - “anchor” -- with dynamic number of columns
    - Each web page may be referenced by different number of other pages
    - E.g. *www.cnn.com* page has two referencing sites
    - Column key is “anchor:<referencing site url>”

Timestamps

- Classic relational model can only store the "current" value of a particular row and its columns
- Bigtable stores multiple versions of a column by design
- Version is indexed by a 64-bit timestamp
  - System time or assigned by client
  - If system time is used, this is equivalent to transaction time
  - Client assigned time can have various meanings
- 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

Web Table with Timestamp

	language	content	anchor	
			“connsi.com”	“my.look.ca”
“com.cnn.www”	“EN” ← t1	“<html>…” ← t1 “<html>…” ← t2 “<html>…” ← t6	“CNN” ← t9	“CNN.com” ← t7

- The multidimensional sorted map concept
  - (**rowkey:string**, **columnKey:string**, **timestampe:int64**) -> value: string
  - Examples:
    - (**“com.cnn.www”, “anchor:consi.com”, t9**) -> “CNN”
    - (**“com.cnn.www”, “language:”, t1**) -> “EN”

Typical APIs

- Data definition API
  - Create/delete table and column families
  - Update table/column family metadata
- Data Manipulation API
  - Write or delete value as specified by rowkey and some column qualifier
  - Look up specific row by row key
  - Scan a short range of rows
  - Support single row transaction

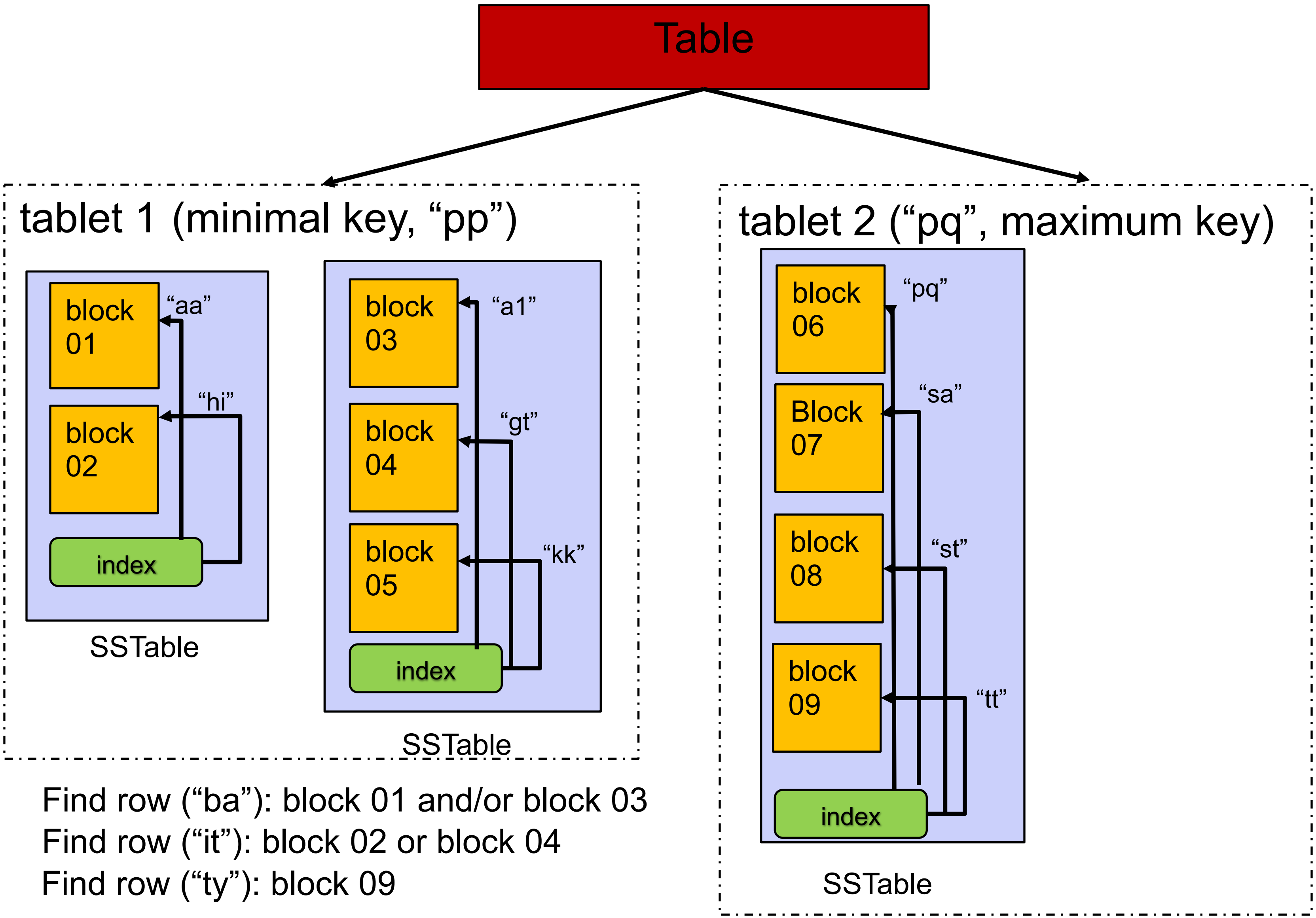
Outline

- Log Structured Merge Tree
- Bigtable Data model
- **Bigtable Architecture**
  - Immutable SSTable file
  - Master-Tablet Server Architecture
  - Chubby Services
  - Tablet Representation and Write/Read Path

Data Storage

- Google File System (GFS)
  - Is used to store actual Bigtable data (log and data files)
  - It provides replication/fault tolerance and other useful features in a cluster environment
- Google SSTable file format
  - Bigtable data are stored internally as SSTable format
    - Sorted String Table
  - Each SSTable consists of
    - Blocks (default 64KB size) to store **ordered** *immutable* map of key value pairs
    - Block index
- The SSTable is stored as GFS files and are replicated

Table-Tablet-SSTable



Find row (“ba”): block 01 and/or block 03
Find row (“it”): block 02 or block 04
Find row (“ty”): block 09

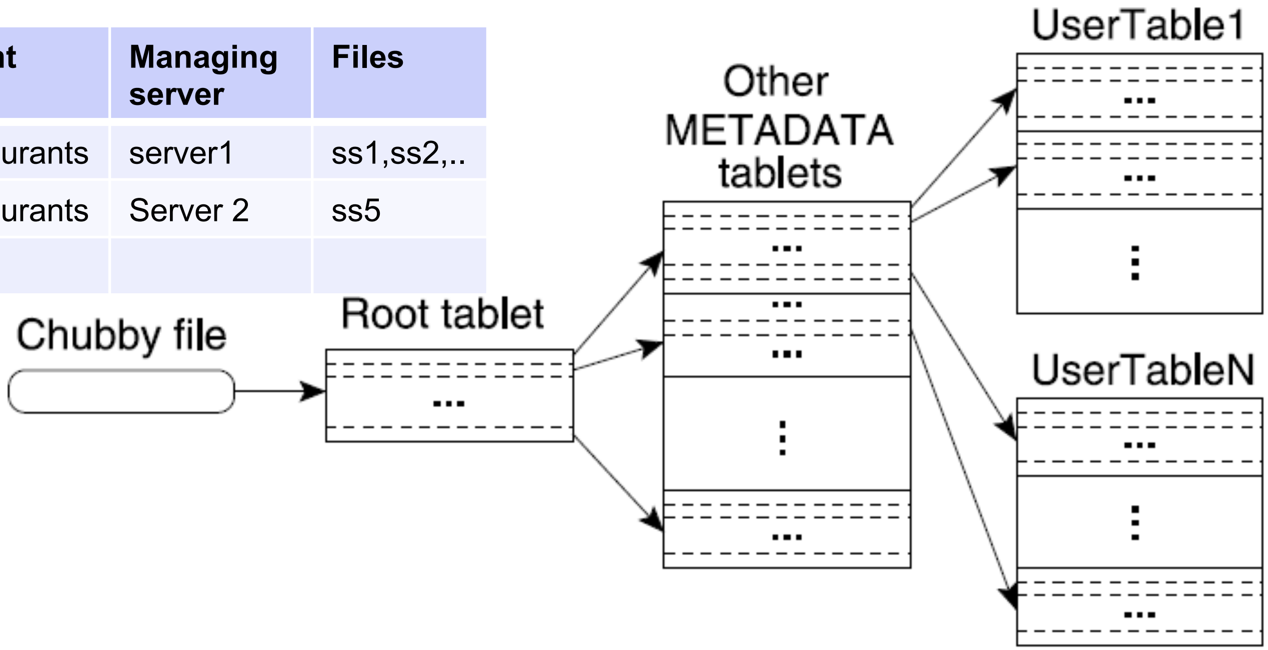
Architecture

- **Many** *tablet servers*
  - 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
- **One** *master server*
  - Assigns tablets to tablet server
  - Balances tablet server load
  - Garbage collection of unneeded files
  - Schema changes (table & column family creation)
  - It is **NOT** in the read/write path
- Client library

Tablet Location

- METADATA table contains the location of all tablets in the cluster
  - It might be very big and split into many tablets
- The location of METADATA tablets is kept in a root tablet
  - This can never be split and its location is stored in Chubby
- Each tablet is <u>assigned</u> to be managed by **ONE** tablet server at a time.
- Both ROOT and METADATA tablets are managed by tablet servers as well

Tablet id	Key range	Parent table	Managing server	Files
1	“c“- “ca”	Restaurants	server1	ss1,ss2,..
2	“cb”-”g”	Restaurants	Server 2	ss5

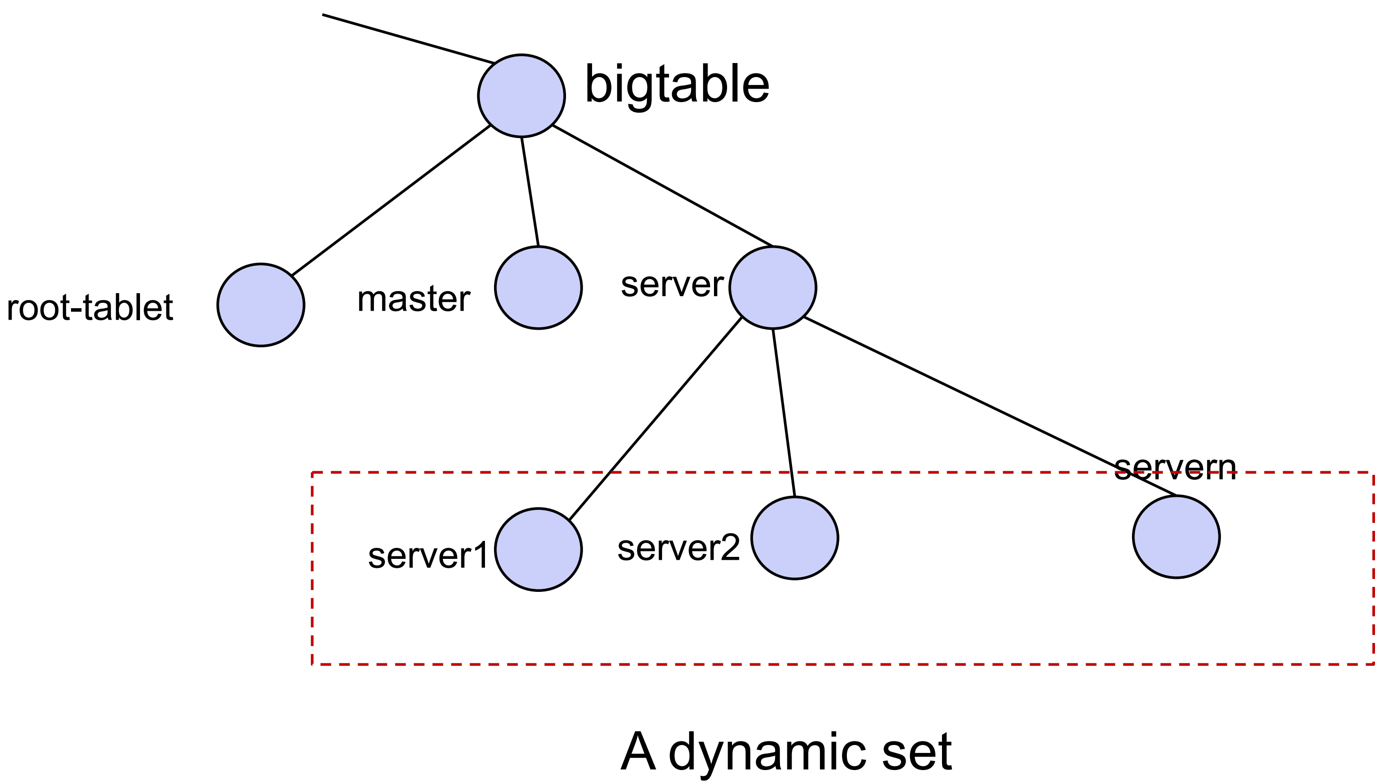


Chubby Services

- Chubby is distributed lock service consists of a small number of nodes (~5)
  - Each is a replica of one another
  - One is acting as the master
  - Paxos is used to ensure majority of the nodes have the latest data
- Usage in Bigtable
  - Ensure there is only one master
  - Keep track of all tablet servers
  - Stores the root table location
  - If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.

Chubby Bigtable File Hierarch Example

- Chubby exports UNIX file system like APIs.
- It allows clients to create directory/file (node) and locks on them
  - Lock has short lease time and needs to be renewed periodically



Chubby and Tablet Servers

- Tablet servers are able to join or leave a running cluster without interfering the normal cluster operation
- Chubby is used to keep track of tablet servers
- Normal handling
  - Each server creates & locks a unique file in *Server Directory* when it starts
  - The lock has short lease and needs to be renewed periodically
  - If a tablet server is scheduled to leave the cluster, it will release its lock
- Error handling
  - A tablet server may lose the lock (e.g. expires)
    - It will stops serving the tablets
    - It will report to master that the lock is lost
    - It will attempt to reclaim the lock if the file still exists, otherwise it kills itself
  - A tablet server may crash and its file become orphaned
    - Master will come to the rescue

Chubby and Master Operation

- Master also obtains an *exclusive master* lock from chubby to ensure there is only one master server
- Master monitors Chubby's *server directory* to find the current list of tablet servers in the cluster
- Master detects the status of tablet servers by periodically asking each server for the status of its lock
- Error handling
  - If tablet server is alive but has no lock or if the tablet server is unreachable
    - The master will contact Chubby to acquire a lock on the orphaned server file and delete it
    - The master also assigns all tablets to other servers
  - If a master cannot contact Chubby to renew its lock, it kills itself

Master Start Up

- When a master is started
  1. It grabs a unique master lock in Chubby
  2. Find out all live servers
  3. Communicate with all servers to find out what tablets they serve
  4. Scan the METADATA table to find the total set of tablets in the cluster
     - May discover tablets that are not assigned
  5. Assign tablets without a server to a new tablet server
- Any cluster has a **root** tablet, in step 3, the master may
  - Find the server that manages the **root** tablet and proceed with step 4
  - Find that the **root** tablet is not assigned to any server, the master will assign it to a server and proceed with step 4

Assigning Tablet to Tablet Server

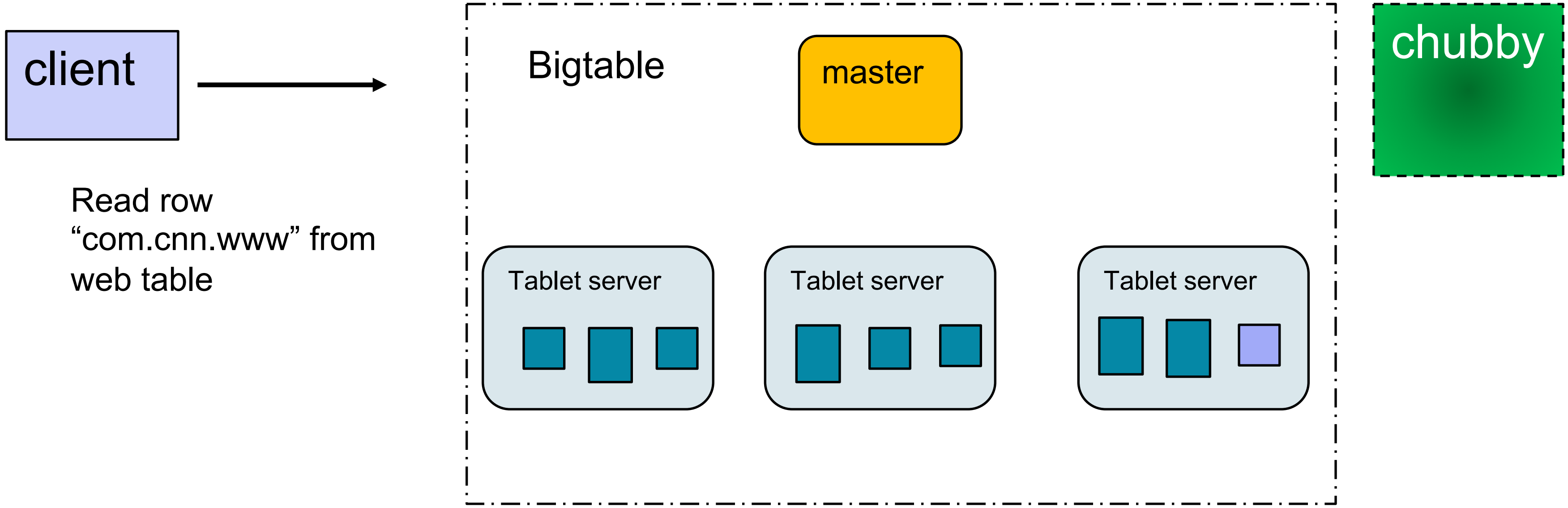
- Scenarios that will trigger tablet assignment
- During start up
  - Master assign tablets to servers to <u>balance the load</u>
- When data changes
  - Tables are created or deleted (master initiates)
  - Two tablets are merged to form one (master initiates)
  - An existing tablet is split into two smaller ones (tablet server initiates)
- When a tablet server is down
  - The tablets it manages need to be assigned to other tablet servers
- When a new tablet server joins
  - The master needs to allocate tablets to it.

Assigning Tablet to Tablet Server (cont'd)

- The assignment is initiated by master sending a **load tablet** request to a tablet server.
- Upon receiving such request, a tablet server performs the following:
  - Scan the METADATA table to find information about this tablet
    - List of SSTable files
    - Log file
  - Read the block indexes in memory
  - Play the log file to reconstruct the memory with all updates are not yet persisted in SSTables

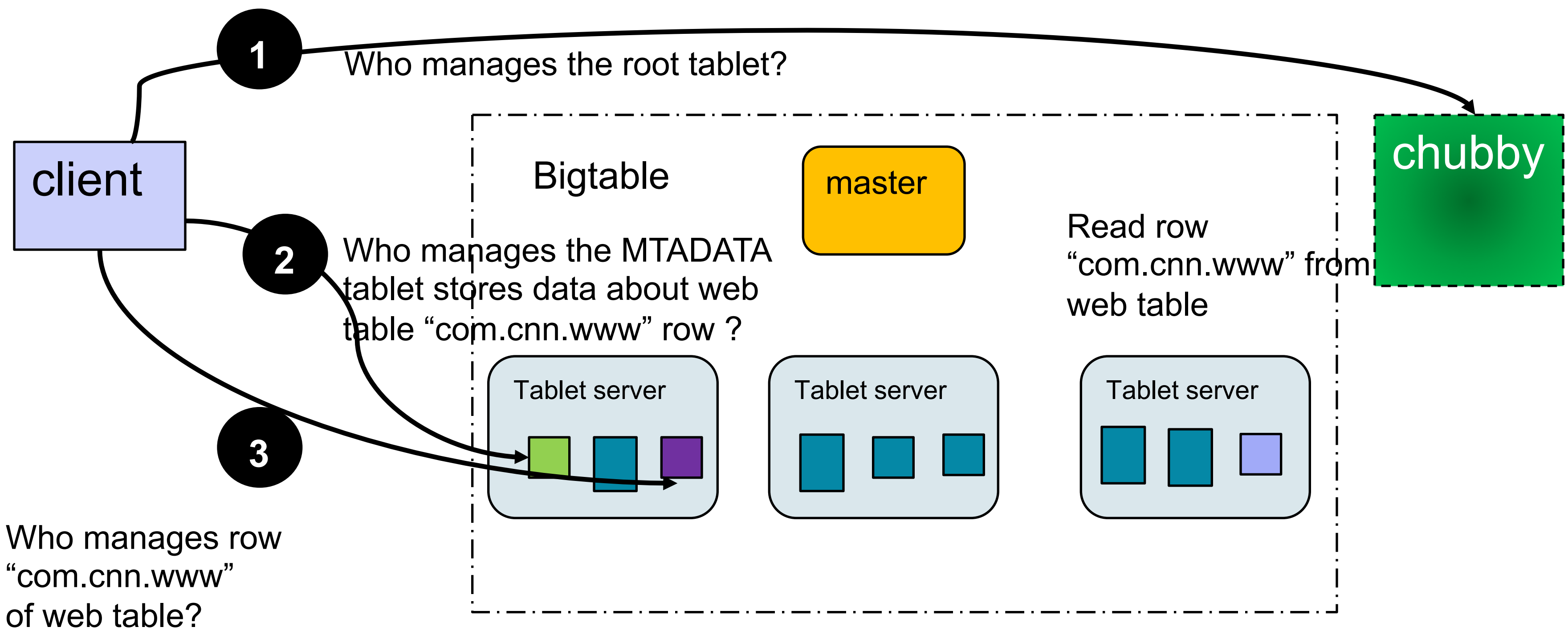
Tablet Serving

- Client read/write request
  - E.g. client wants to read the row corresponding to “com.cnn.www” from the web table
- Steps
  - Find the *tablet location*, the table server that serves the tablet
  - Contact the tablet server to perform the read/write request

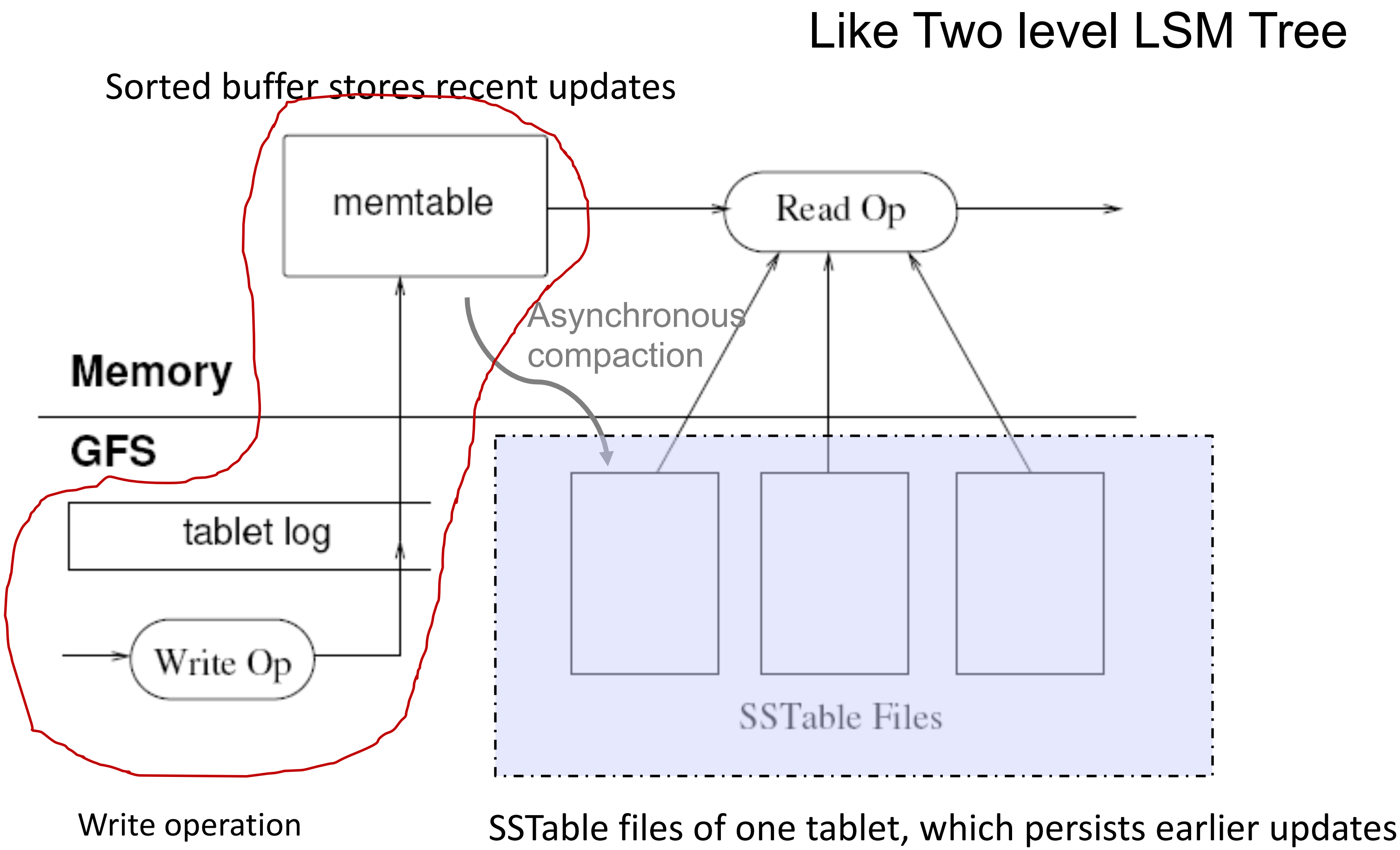


Find the tablet server

- If the client is requesting the data for first time
  - One round trip from chubby to find the root tablet's location
  - One round trip to the tablet server manages the root tablet
  - One round trip to the tablet server manages the METADATA tablet
- The client caches the tablet location for later use



Tablet Representation



Tablet Representation Implications

- A tablet server *manages* many tablets
  - Its memory contains latest updates of those tablets
  - BUT, the actual persisted data of those tablets might not be stored in this tablet server
    - Logs and SSTable Files are managed by the underlying file system GFS
    - GFS might replicate the files in any server
- Bigtable system is not responsible for actual file replication and placement
- The separation of concern simplifies the design

Write Path

- A write operation may insert new data, update or delete existing data
- The client sends write operation directly to the tablet server
  - The operation is checked for syntax and authorization
  - The operation is written to the **commit log**
  - The actual mutation content is inserted in the **memtable**
    - Deleted data will have a special tombstone entry/marker
- The only disk operation involved in write path is to append update to commit log

Compactions

- After many write operations, the size of memtable increases
- When memtable size reaches a threshold
  - The current one is frozen and converted to an SSTable and written to GFS
  - A new memtable is created to accept new updates
  - This is called **minor compaction**
- Why minor compaction
  - Memory management of tablet server
  - Reduce the size of active log entries
    - Minor compaction persists the updates on disk
    - Log entries reflecting those updates are no longer required

Compactions (cont'd)

- Every **minor compaction** creates a new SSTable
  - A tablet may contain many SSTable with overlapping key ranges
- **Merging compaction** happens periodically to merge a few SSTables and the current memtable content into a new SSTable
- **Major compaction** write all SSTable contents into a single SSTable. It will permanently remove the deleted data.

Compaction Process (t1-t5)

	language	content	anchor	
“cnn”	“language:” -> “EN” ← t1	“content:” -> “<html> …” ← t1 “content:” -> “<html> …” ← t20	“anchor:cnnsi.com” -> “CNN” ← t6	“anchor:my.look.ca” -> “CNN.com” ← t13
“zdnet”	“language:” -> “EN” ← t4	“content:” -> “<html> …” ← t4	“anchor:slashdot.com” -> “zdnet” ← t8	

- Suppose a minor compaction happens at t5

Memstore

```
(“cnn”, “content:”,t1) -> “<html..”
(“cnn”,“language:”,t1) -> “EN”

(“zdnet”,”content:”,t4) -> “<html..”
(“zdnet”,“language:”,t4) -> “EN”
```

SSTable File 1

Compaction Process (t6-t14)

	language	content	anchor	
"cnn"	"language:" -> "EN" ← t1	"content:" -> "<html>…" ← t1 "content:" -> "<html> …" ← t20	"anchor:cnnsi.com" -> "CNN" ← t6	"anchor:my.look.ca" -> "CNN.com" ← t13
"zdnet"	"language:" -> "EN" ← t4	"content:" -> "<html> …" ← t4	"anchor:slashdot.com" -> "zdnet" ← t8	

- Suppose another minor compaction happens at t14

Memstore

```
("cnn","anchor:cnnsi.com",t6) >"CNN"
("cnn","anchor:my.look.ca:",t13)->"CNN.com"
("zdnet","anchor:Slashdot.com",t8)->"zdnet"
```

SSTable File 1

```
("cnn", "content:",t1) -> "<html.."
("cnn","language:",t1) -> "EN"
("zdnet","content:",t4) -> "<html.."
("zdnet","language:",t4) -> "EN"
```

SSTable File 2

Compaction Process (t15)

- Assume a merging compact happens at **t15**

SSTable File 1

```
(“cnn”, “content:”,t1) -> “<html..”
(“cnn”,“language:”,t1) -> “EN”
(“zdnet”,”content:”,t4) -> “<html..”
(“zdnet”,“language:”,t4) -> “EN”
```

SSTable File 2

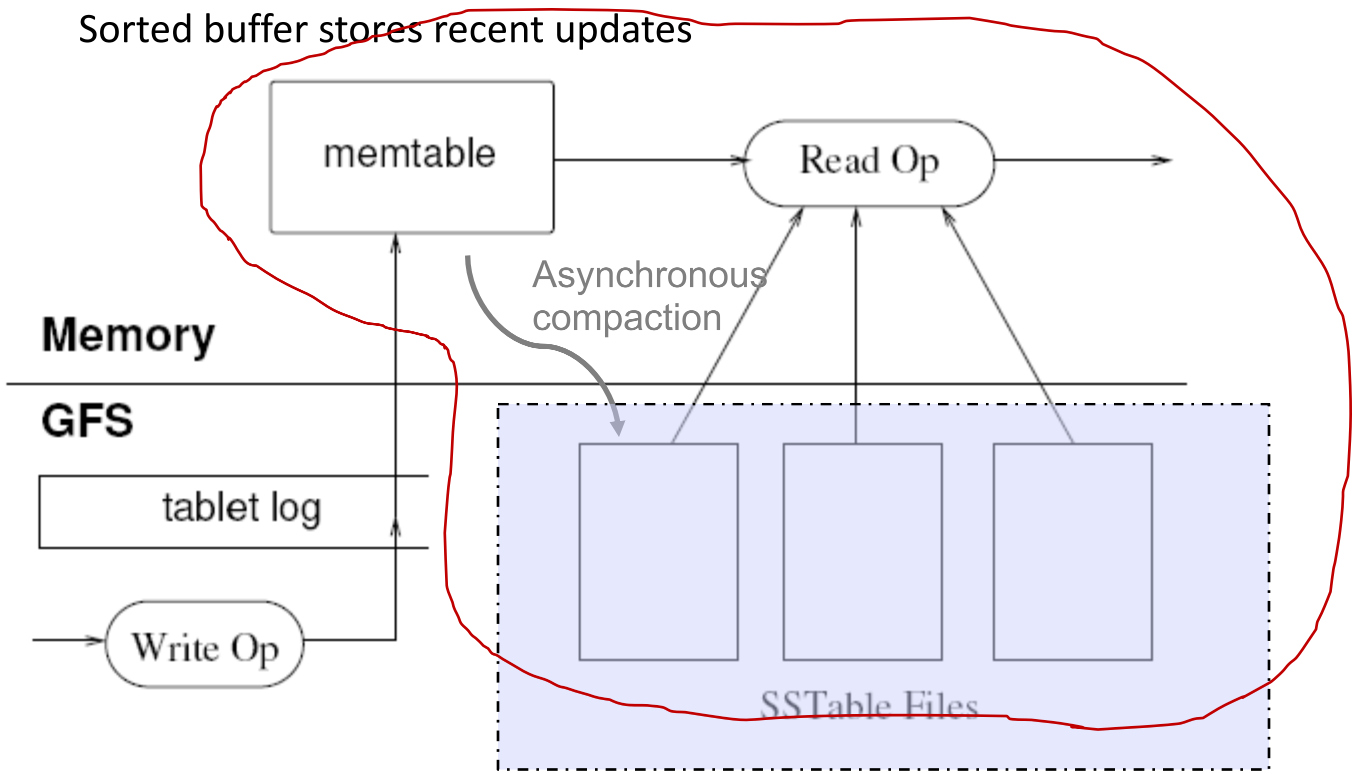
```
(“cnn”,“anchor:cnnsi.com”,t6) >“CNN”
(“cnn”,“anchor:my.look.ca:”,t13)->“CNN.com”
(“zdnet”,“anchor:Slashdot.com”,t8)->“zdnet”
```

SSTable File 3

```
(“cnn”,“anchor:cnnsi.com”,t6) >“CNN”
(“cnn”,“anchor:my.look.ca:”,t13)->“CNN.com”
(“cnn”, “content:”,t1) -> “<html..”
(“cnn”,“language:”,t1) -> “EN”
(“zdnet”,“anchor:Slashdot.com”,t8)->“zdnet”
(“zdnet”,”content:”,t4) -> “<html..”
(“zdnet”,“language:”,t4) -> “EN”
```

Read Path

Sorted buffer stores recent updates



Read Path

- The client sends read operation directly to the tablet server
  - The operation is check for syntax and authorization
  - Both memory and disk maybe involved to obtain the data
- What are kept in memory
  - Most recent updates in memtable (sorted by key)
  - Block indexes of SSTable files
- What are kept in disk
  - Earlier updates persisted in one or many SSTable files
- How does tablet server find the data
  - Check if the memtable contains partial data, or special mark indicating certain data is deleted
  - Check the index to find the block(s) that may contain partial data
  - Load the block and extract the data if there is any
  - Combine the data from memtable and disk block to obtain the final result

References

- O'Neil, P., et al. (1996). "The log-structured merge-tree (LSM-tree)." Acta Informatica **33**(4): 351-385.
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, **Bigtable: A Distributed Storage System for Structured Data, OSDI**'06: In Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI'06), Seattle, WA, 2006