Structured Big Data 1: Google Bigtable & HBase

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

Outline

- Problems of big data processing with Hadoop MapReduce
- Structured big data processing
- Traditional RDBMS
- ACID vs BASE
- Distributed DB
- Bigtable
  - Motivations, data model, system architecture, API, implementation, refinement
- Hbase
- CAP theorem

Problems of BD Processing

- Hadoop MapReduce is simple and powerful but:
  - The one-input data format (key-value pairs) and two-stage dataflow computing are extremely rigid.
  - Custom code has to be written for even the most common operations (e.g., projection and filtering)
- Programmers could be unfamiliar with the MapReduce and would prefer to use SQL-like lang
- Performing tasks with a different dataflow (e.g., joins or *n* stages) would require implementing inelegant workarounds
- Hadoop MR is not good for interactive queries

Structured Big Data Processing

- This lecture discuss various solutions on adding SQL flavor on top of the Big Data platforms for processing large-scale structured data.
- Starting with the structured data store Google Bigtable.
- Then discuss the open source counterpart Apache Hbase.
- Move toward NoSQL.

Traditional DBMS

- Mostly based on relational model (tables, tuples, attributes)
- Well-defined schema
- Support relational operators (SELECT, PROJECT, JOIN, …)
- SQL language
- Transaction management
- ACID properties

ACID Properties

- Atomicity
  - either all the operations of a transaction are executed or none of them are (all-or-nothing)
- Consistency
  - the database is in a legal state before and after a transaction
- Isolation
- Durability

ACID Properties

- Atomicity
- Consistency
- Isolation
  - the effects of one transaction on the database are isolated from other transactions even under concurrent execution
- Durability
  - the effects of successfully completed (i.e., committed) transactions endure subsequent failures

Benefits of RDBMS

- High level semantics
  - Easy to program
  - Programmers are more familiar with
  - (Multi-row) transactions
- Lots of mature commercial implementations
  - MySQL, PostgreSQL, MSSQL.....
- Optimizations makes them really fast
  - But only under small scale of data

Problems of RDBMS on Large Scale Data

- Most important of all, current implementations lack, or only come with limited support of distributed deployment
- Not very feasible when it comes to BIG data.
  - Especially when it come to scalability
- ACID properties are too strong (next slide)

ACID vs BASE

- ACID properties seem indispensable
- They are incompatible with availability, scalability and performance requirements in very large systems.
- An alternative to ACID is **BASE**:
  - **B**asic **A**vailability
  - **S**oft-state
  - **E**ventual consistency

CAP Theorem

- Eric Brewer (Brewer's Theorem): It is impossible for a distributed system to simultaneously provide all three of the following guarantees:
  - ***Consistency***
  - ***Availability***
  - ***Partition tolerance***
- (more on this later)

Distributed Database

- Remind:Transaction
  - A unit of consistent and atomic execution against the database.
- Termination protocol
  - A protocol by which individual sites can decide how to terminate a particular transaction when they cannot communicate with other sites where the transaction executes.
- Distributed DBMS, Concurrency control algorithm, Distributed Locking, Logging protocol, One-copy equivalence, Query processing, Query optimization, Quorum-based voting algorithm, Read-once, write-all protocol, Serializability, Transparency, Two-phase commit, Two-phase locking

BigTable: Motivations

- Consider Google …
  - Lots of (semi-)structured data
    - Copies of the web, satellite data, user data, geographic data, email and USENET, Subversion backing store
  - Millions of machines
  - Different projects/applications
  - Hundreds of millions of users
  - Many incoming requests (thousands of q/sec)
  - 100TB+ of satellite image data
- Need both offline data processing and online serving

Why not a DBMS?

- **Few DBMS's support the requisite scale**
  - Required DB with wide scalability, wide applicability, high performance and high availability
- **Couldn't afford it if there was one**
  - Most DBMSs require very expensive infrastructure
- **DBMSs provide more than Google needs**
  - E.g., full transactions, SQL
- **Google has highly optimized lower-level systems that could be exploited**
  - GFS, Chubby(distributed lock service), MapReduce, Job scheduling

BigTable: Goals

- Wide applicability
  - Can be used by many Google products and projects
  - Often want to examine data changes over time, e.g., Contents of a web page over multiple crawls
  - Both throughput-oriented batch-processing jobs and latency-sensitive serving of data to end users
- Scalability
  - Handful to thousands of servers, hundreds of TB to PB
- High performance
  - Millions of ops per second
- High availability
  - Want access to most current data at any time

What is a BigTable?

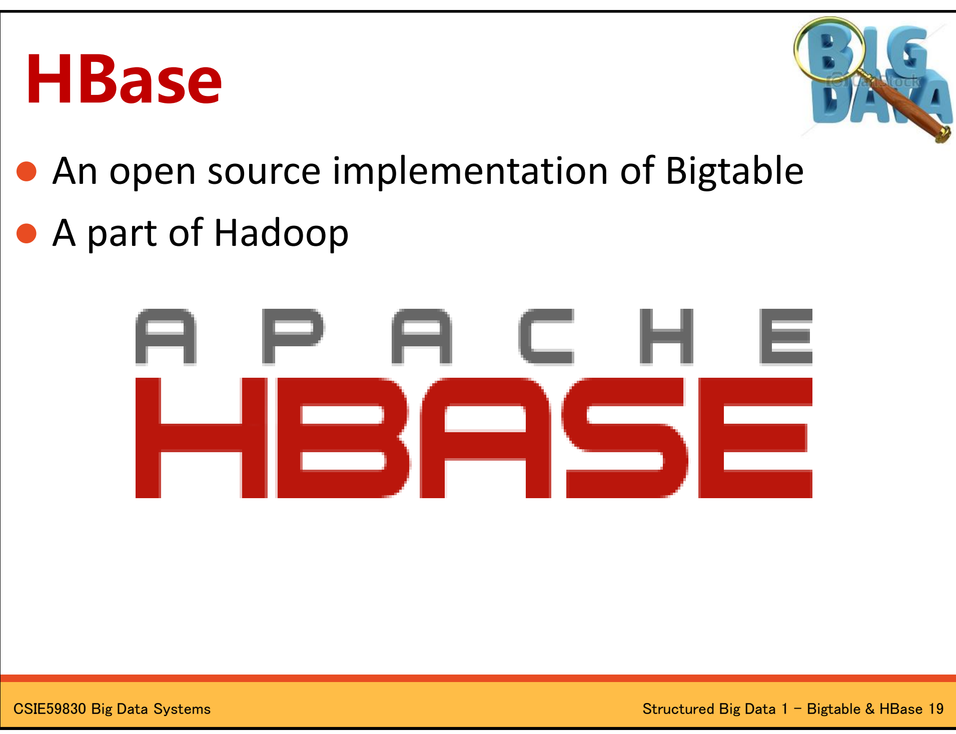
- "A BigTable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, a column key, and a timestamp; each value in the map is an uninterpreted array of bytes."
  - Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In 7th OSDI 2006.

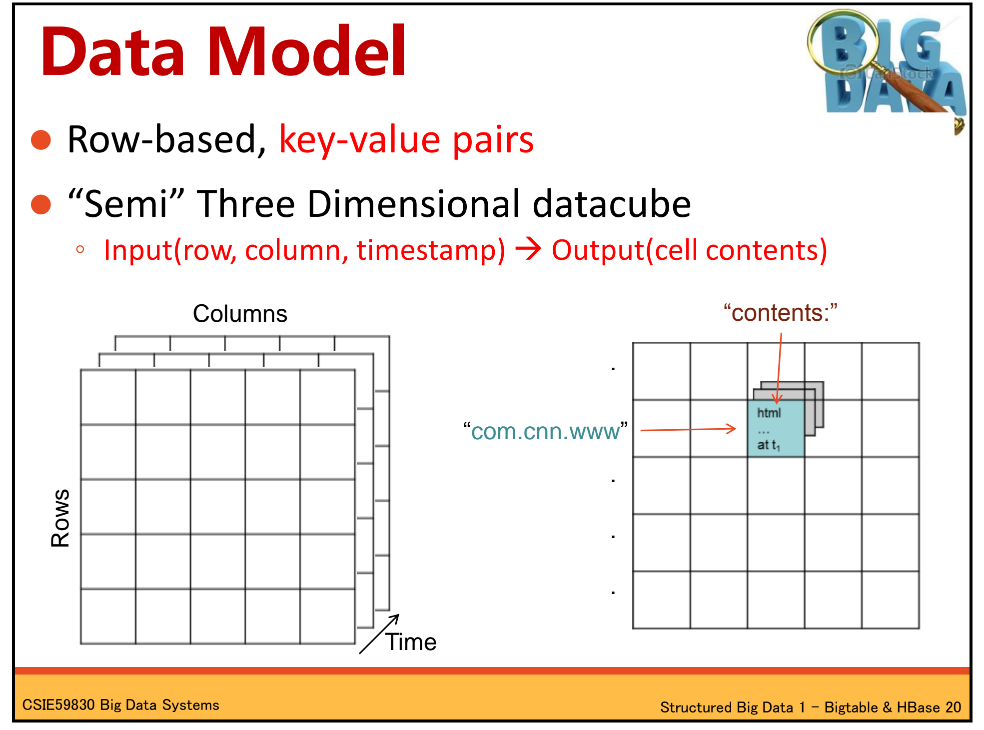
BigTable: Introduction

- A sparse, distributed, persistent multidimensional sorted map
  - With an interesting data model
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

Relative to DBMS, BigTable provides …

- Simplified data retrieval mechanism
  - A map
  - <Row, Column, Timestamp> -> string
  - No relational operators
- Atomic updates only possible at row level
- Arbitrary number of columns per row
- Arbitrary data type for each column
- Designed for Google’s application set
- Provides extremely large scale (data, throughput) at extremely small cost


HBase
An open source implementation of Bigtable
A part of Hadoop
APACHE
HBASE
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 19


Data Model
Row-based, key-value pairs
"Semi" Three Dimensional datacube
Input(row, column, timestamp) → Output(cell contents)
Columns
Rows
Time
"contents:"
"com.cnn.www"
html
...
at t1
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 20

Data Model: Rows

- Row keys are arbitrary strings up to 64KB
- Row is the unit of transactional consistency
  - Every read or write of data under a single row is atomic
  - Multi-row atomicity not guaranteed
- Identified and sorted in lexicographic order by row keys
- Rows with consecutive keys (Row Range) are grouped together as "*tablets*".
  - Unit of distribution and load-balancing
  - reads of short row ranges are efficient and typically require communication with only a small number of machines

Data Model: Columns

- Provide schema-like semantic
- Column keys are grouped into sets called "*column families*", which form the unit of access control.
- Data stored under a column family is usually of the same type (easier to be compressed together)
- A column family must be created before data can be stored in a column key
  - After a family has been created, any column key within the family can be used for queries
- Column key is named using: ***family:qualifier***
- Access control and disk/memory accounting are performed at column family level
- Managed by the ***Chubby*** lock service

Data Model: Timestamps

- Each cell can contain multiple versions of data, each indexed by *timestamp* (called *version* in Hbase)
- Timestamps are 64-bit integers
- Assigned by:
  - **Bigtable**: real-time in microseconds
  - **Client application**: when unique timestamps are a necessity
- Data is stored in decreasing timestamp order, so that most recent data is easily accessed
  - Application specifies how many versions (n) or how new enough (last 7 days) items to be maintained in a cell
  - Bigtable garbage collects obsolete versions

Data Model Example

Example: Zoo

Data Model Example

Example: Zoo

row key col. key timestamp

Data Model

Example: Zoo

row key col. key timestamp

- (zebras, length, 2006) --> 7 ft
- (zebras, weight, 2007) --> 600 lbs
- (zebras, weight, 2006) --> 620 lbs

Data Model

Example: Zoo

Each key is sorted in Lexicographic order

row key col. key timestamp

- (zebras, length, 2006) --> 7 ft
- (zebras, weight, 2007) --> 600 lbs
- (zebras, weight, 2006) --> 620 lbs

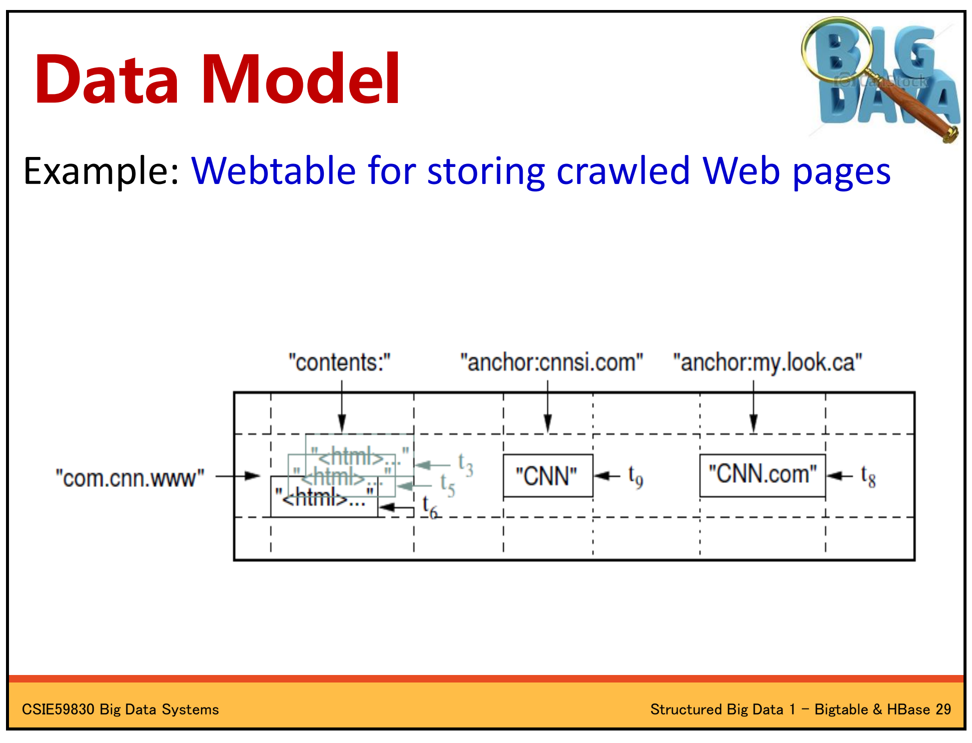
Data Model

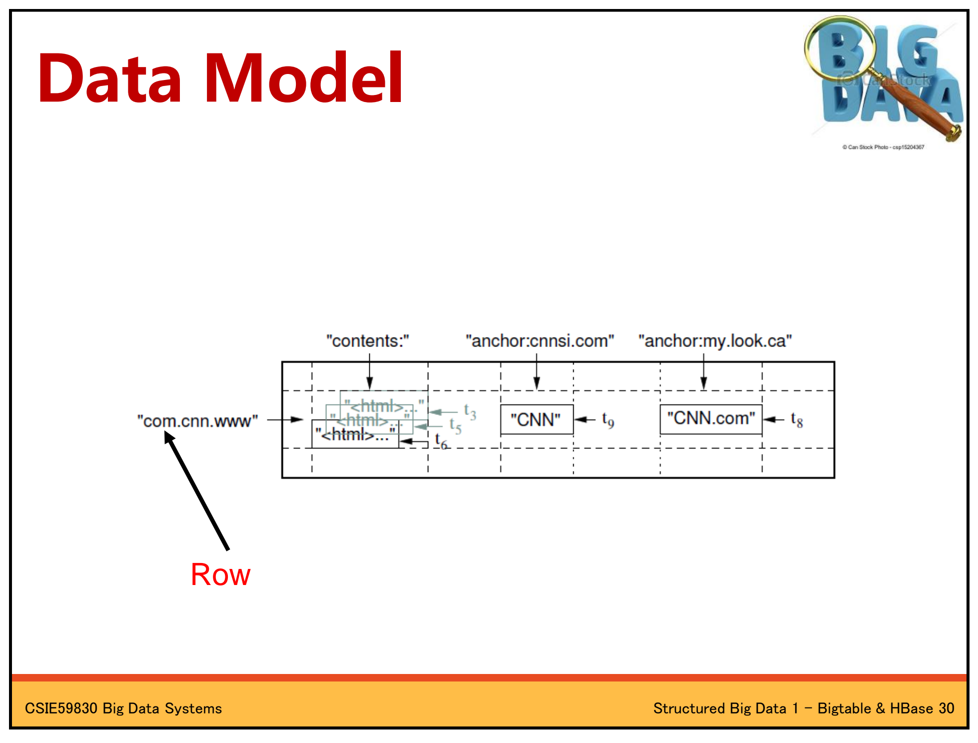
Example: Zoo

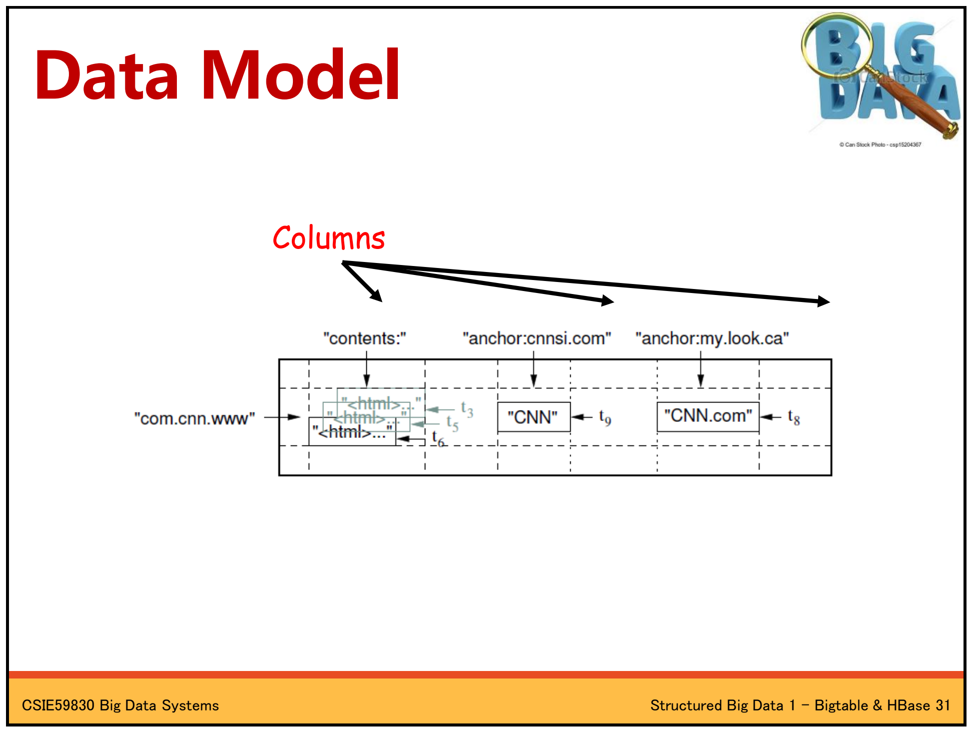
Timestamp ordering is defined as "most recent appears first"

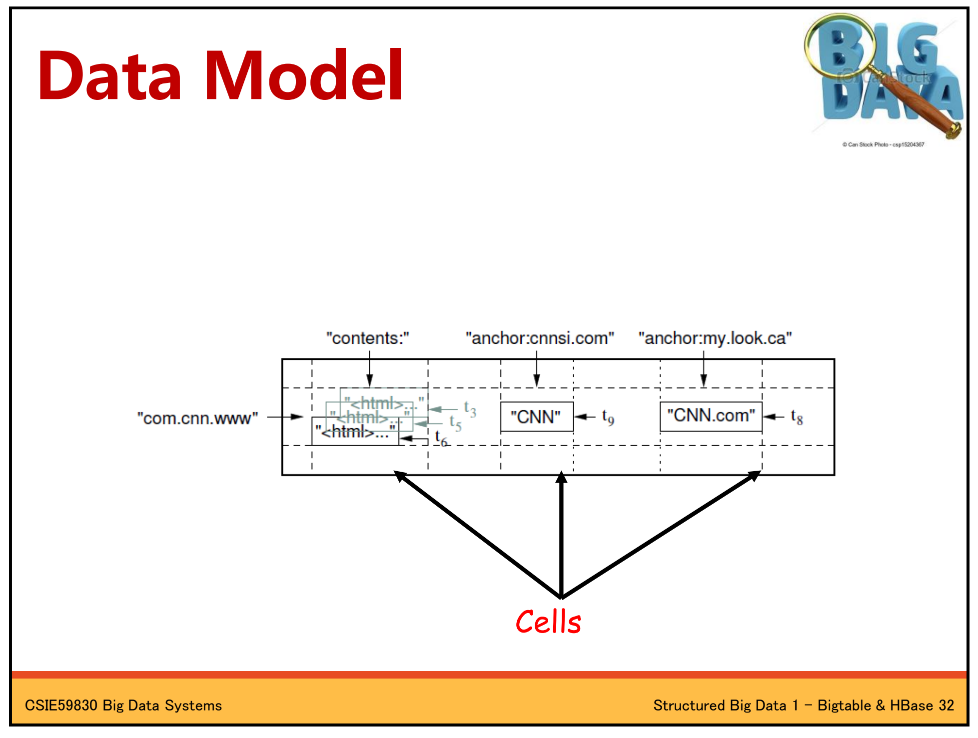
row key col. key timestamp

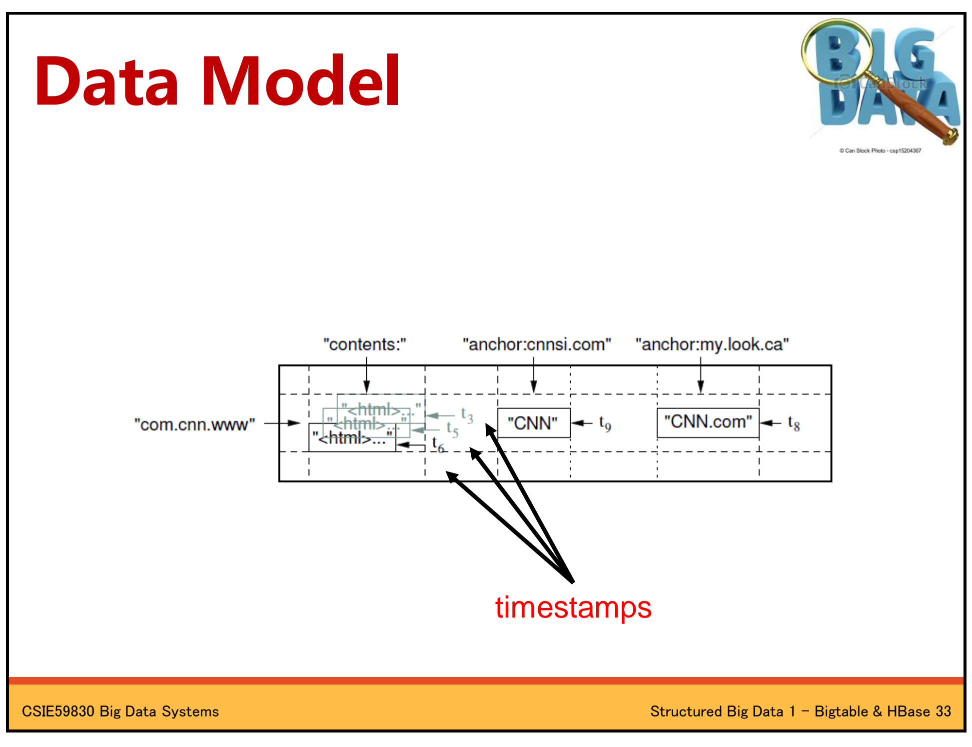
- (zebras, length, 2006) --> 7 ft
- (zebras, weight, 2007) --> 600 lbs
- (zebras, weight, 2006) --> 620 lbs

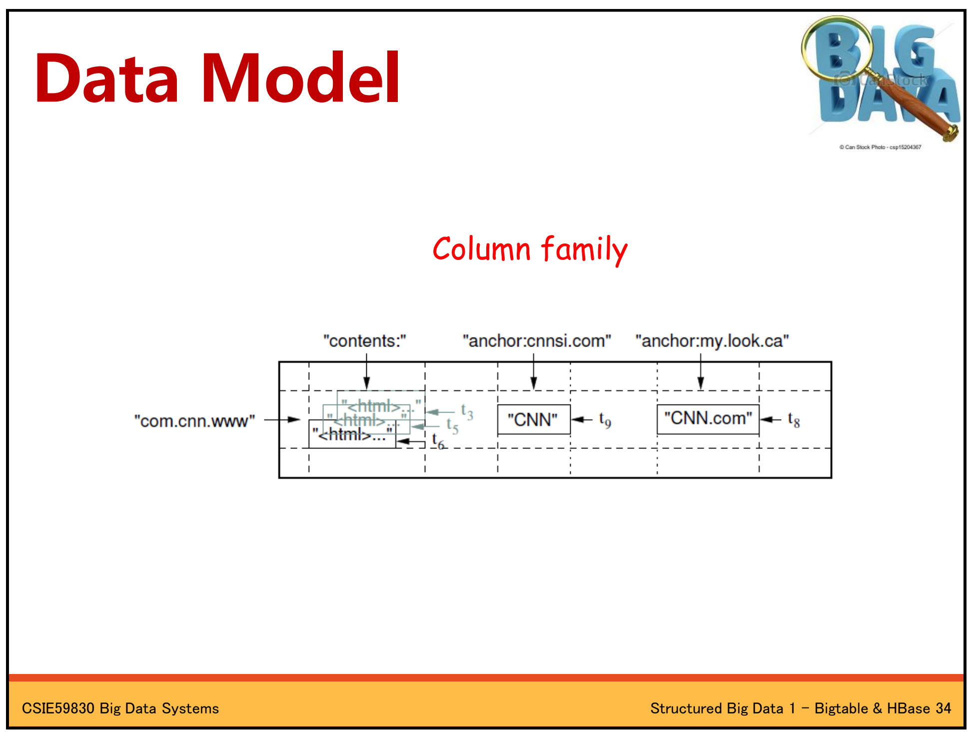

Data Model
Example: Webtable for storing crawled Web pages
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 29

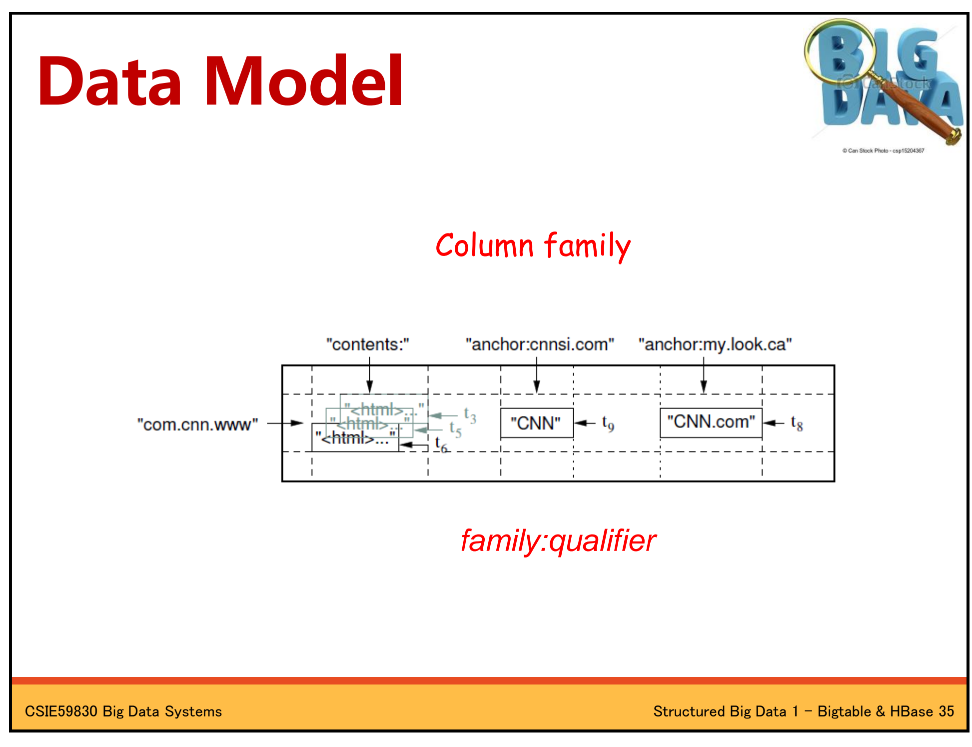

Data Model
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
Row
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 30

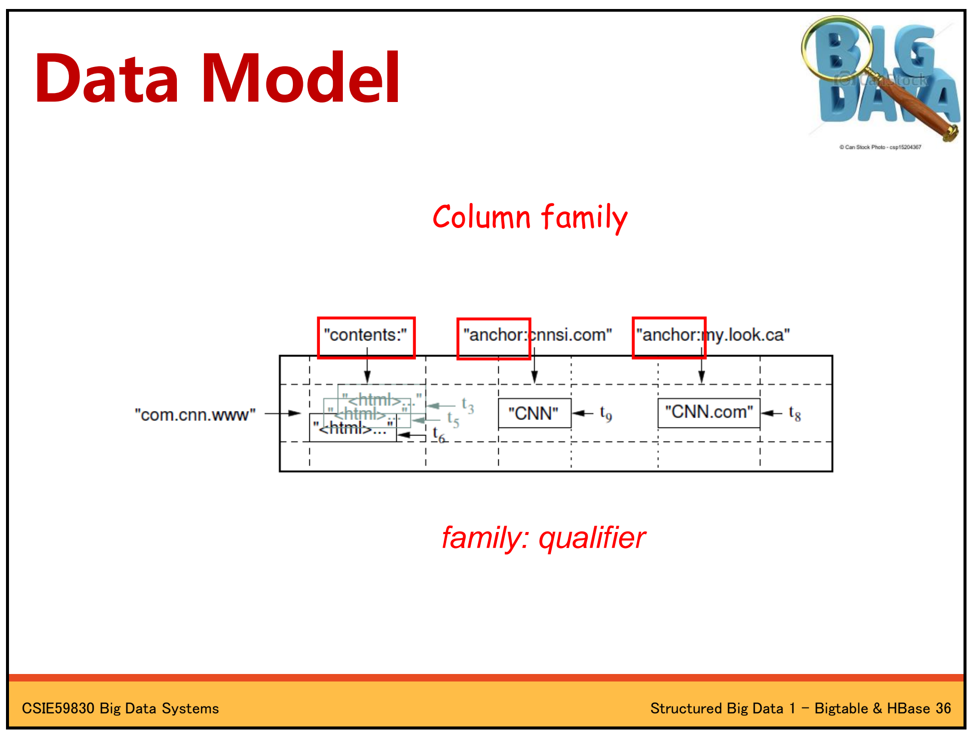

Data Model
Columns
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 31


Data Model
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
Cells
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 32


Data Model
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
t_3
t_5
t_6
t_9
t_8
timestamps
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 33


Data Model
Column family
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
t_3
t_5
t_6
t_9
t_8
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 34


Data Model
Column family
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
family:qualifier
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 35


Data Model
Column family
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"CNN"
"CNN.com"
family: qualifier
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 36

Bigtable API

- Bigtable APIs provide functions for:
  - Creating/deleting tables, column families
  - Changing cluster, table and column family metadata such as access control rights
  - Support of single row transactions
  - Allowing cells to be used as integer counters
  - Executing client supplied scripts in the address space of servers

Bigtable API

- Write API
  - Write or delete different granularities up to row
  - Applied atomicity within 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);
```

Bigtable API

- Read API
  - selection by a combination of row, column or timestamp ranges

```
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());
}
```


Google Applications using BigTable

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Building Blocks

- On top of Google File System (vs HDFS)
  - stores persistent data
- Scheduler (in-house):
  - Schedule Bigtable jobs
- Chubby (vs ZooKeeper)
  - As synchronization service
- MapReduce: not a building block, but uses Bigtable / HBase heavily

Building Blocks: GFS

- Bigtable uses the distributed Google File System (GFS) to store log and data files
- The Google **SSTable** file format is used internally to store Bigtable data
- An SSTable provides a persistent , ordered immutable map from keys to values
  - Operations are provided to look up the value associated with a specified key, and to iterate over all key/value pairs in a specified key range

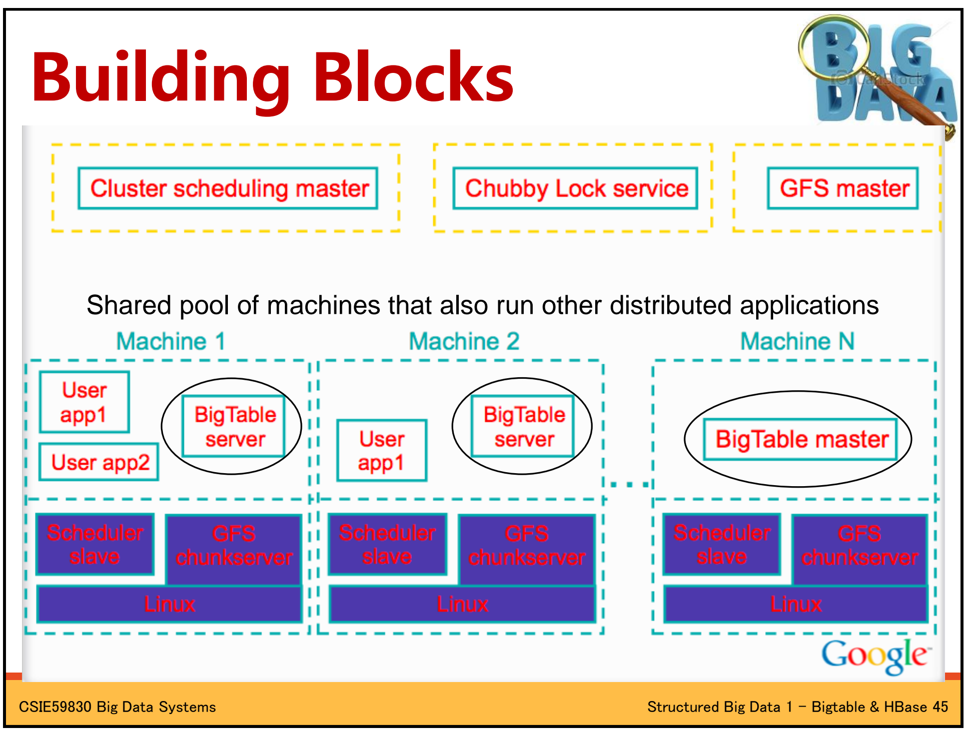
Building Blocks: Chubby

- Bigtable relies on a highly-available and persistent distributed lock service called Chubby
- Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock
  - Consists of 5 active replicas, one replica is the master and serves requests
  - Service is functional when majority of the replicas are running and in communication with one another – when there is a quorum

BigTable and Chubby

- Bigtable uses Chubby to:
  - Ensure there is at most one active master at a time,
  - Store the bootstrap location of Bigtable data (Root tablet),
  - Discover tablet servers and finalize tablet server deaths,
  - Store Bigtable schema information (column family information),
  - Store access control list.
- If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.

Building Blocks



Organization

- A Bigtable cluster stores tables
- Each table consists of tablets
  - Initially each table consists of one tablet
  - As a table grows it is automatically split into multiple tablets
- Tablets are assigned to tablet servers
  - Multiple tablets per server.
  - Each tablet is 100-200 MB
  - Each tablet lives at only one server

System Architecture:Tablet

- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions

System Architecture: Tablet

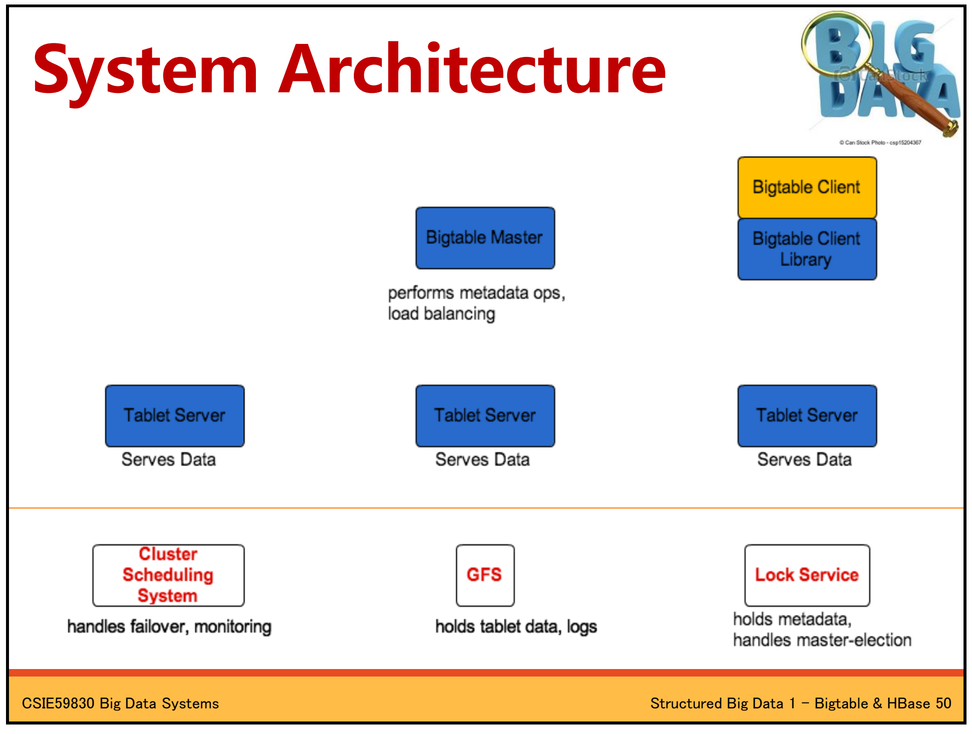
Table
Tablet
Tablet
Tablet
Tablet
...

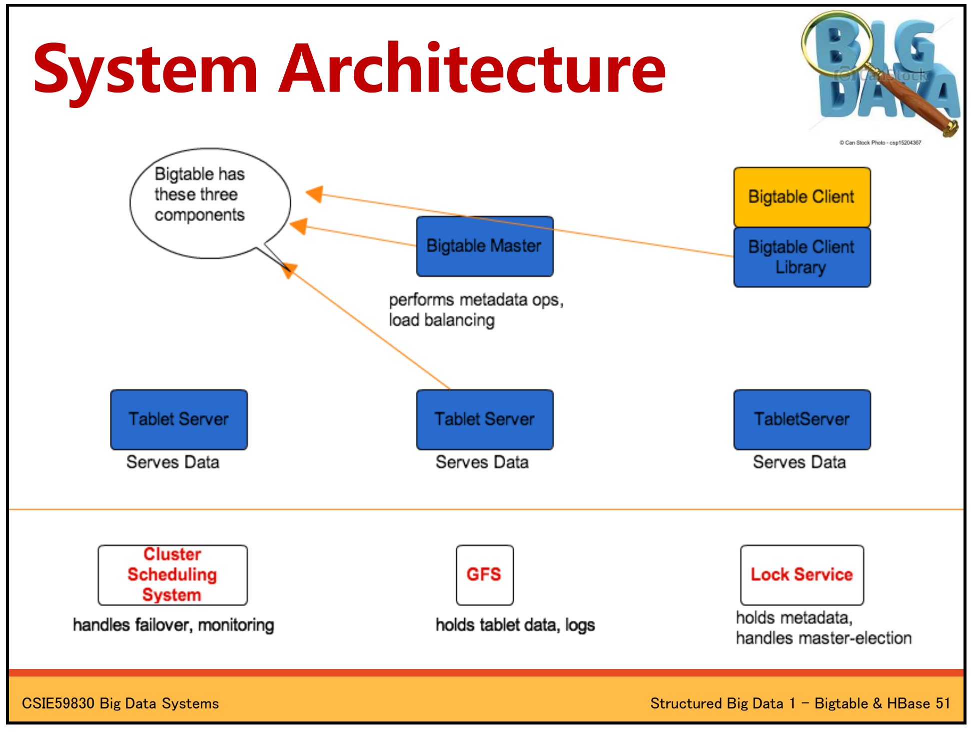
- Dynamic fragmentation of rows
  - Unit of load balancing
  - Distributed over tablet servers
  - Tablets split and merge
    - automatically based on size and load or manually
  - Clients can choose row keys to achieve locality

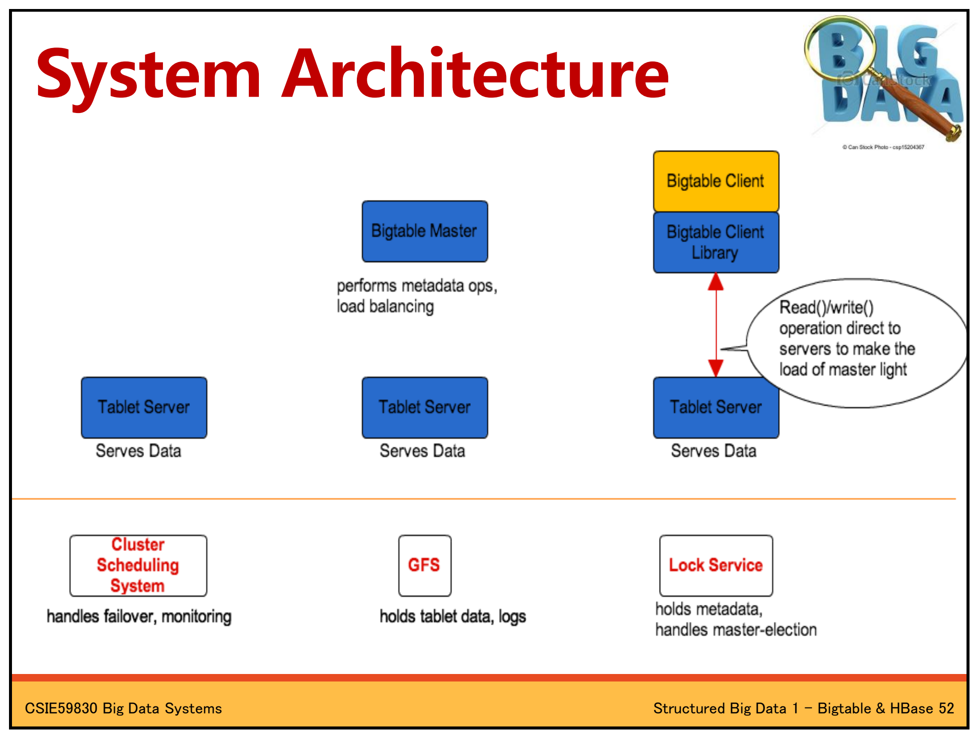
Where is my Tablets?

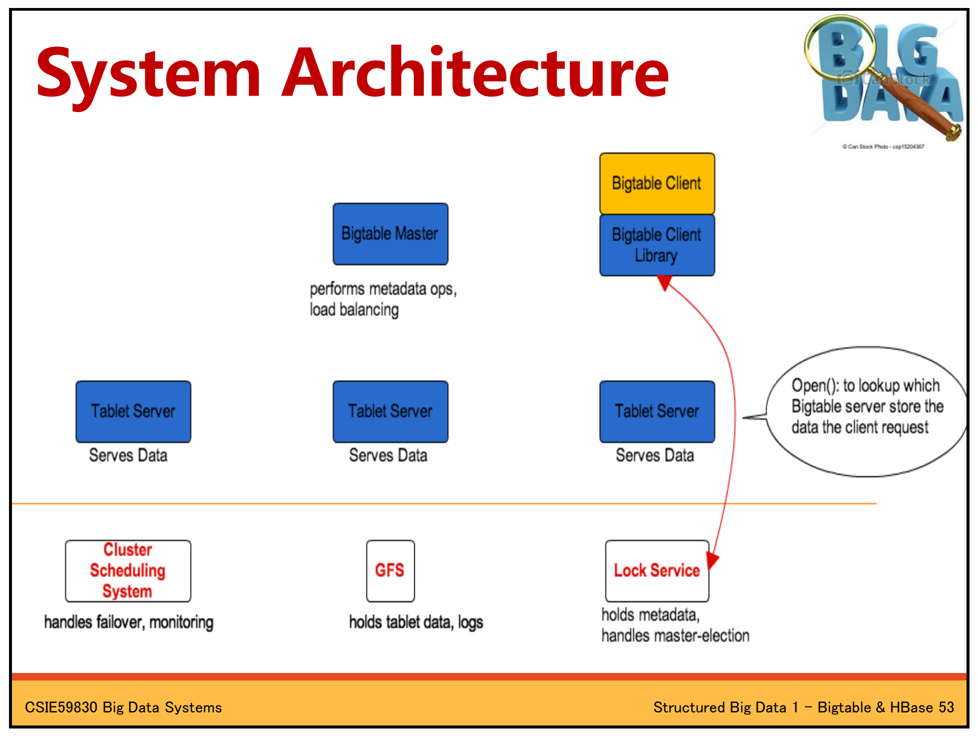
- Question: given a row, how does a client find the right tablet server?
  - Tablet server location is *ip:port*
  - Need to find tablet whose row range covers the target row
  - One approach: could use the BigTable master
    - Central server almost certainly would be bottleneck in large system
    - Instead: store tablet location info in special tablets similar to a B+ tree
    - We'll talk about this later

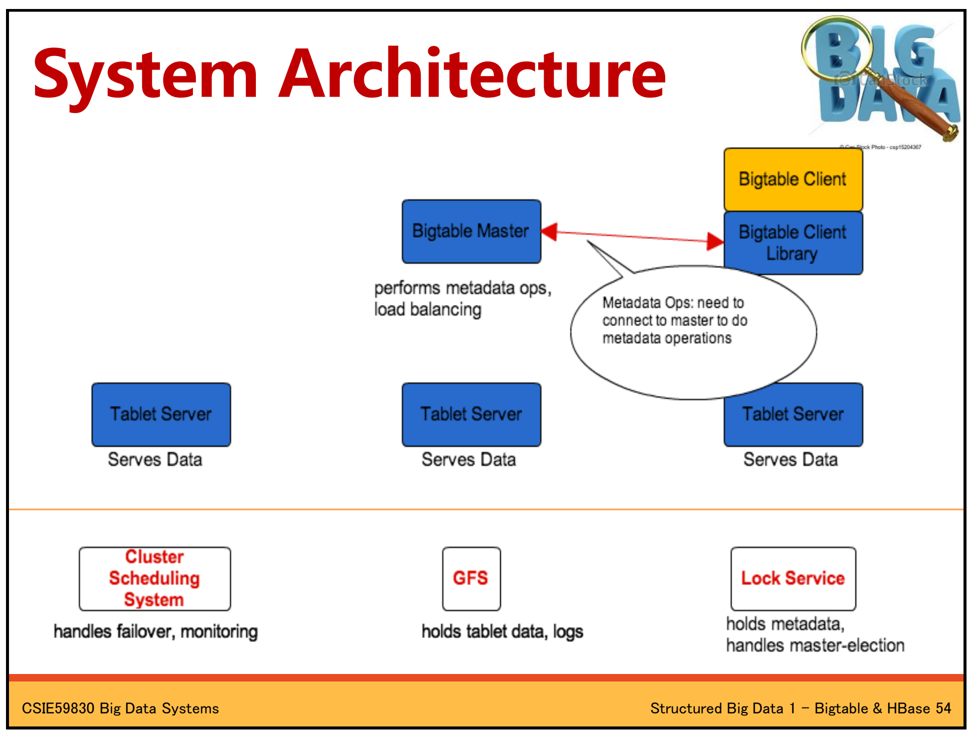
System Architecture




System Architecture
Bigtable has these three components
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Tablet Server
Serves Data
Tablet Server
Serves Data
TabletServer
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 51


System Architecture
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Read()/write() operation direct to servers to make the load of master light
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 52


System Architecture
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Open(): to lookup which Bigtable server store the data the client request
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 53


System Architecture
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Metadata Ops: need to connect to master to do metadata operations
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 54

Implementation

- Tablet Location
- Tablet Serving
- Compaction

Tablet Location

- Remind: Where is my Tablets?
- Question: given a row, how does a client find the right tablet server?
  - Tablet server location is *ip:port*
  - 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 tablet location info in special tablets similar to a B+ tree

Finding Tablet Location

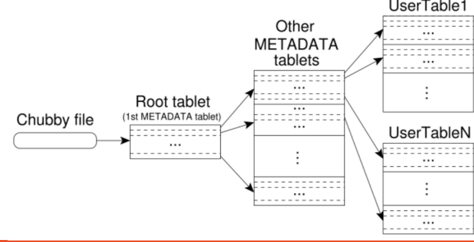
- Client caches tablet locations.
- In case if it does not know, it has to make three network round-trips in case cache is empty and up to six round trips in case cache is stale
- Tablet locations are stored in memory, so no GFS accesses are required

Tablet Location

- A 3-level hierarchy analogous to that of a B+-tree to store tablet location information :
  - A file stored in chubby contains location of the root tablet
  - Root tablet contains location of *Metadata tablets*
    - The root tablet never splits
  - Each metadata tablet contains the locations of a set of user tablets
- Client reads the Chubby file that points to the root tablet
  - This starts the location process
- Client library caches tablet locations
  - Moves up the hierarchy if location N/A

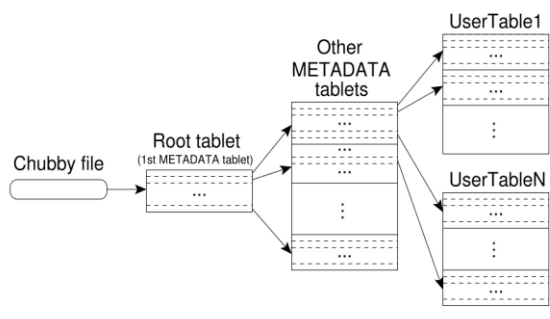
Metadata Tablets

- 3-level B+-tree like scheme for tablets
  - 1st level: Chubby, points to root tablet
  - 2nd level: Root tablet data points to appropriate METADATA tablet
  - 3rd level: METADATA tablets point to data tablets
- METADATA tablets can be split when necessary
- Root tablet never splits so number of levels is fixed



Size Analysis

- Each metadata row stores ~ 1KB of data,
- With 128 MB tablets, the three level store addresses 2^{34} tablets (2^{61} bytes in 128 MB tablets).
- Approaches a Zetabyte (million Petabytes).

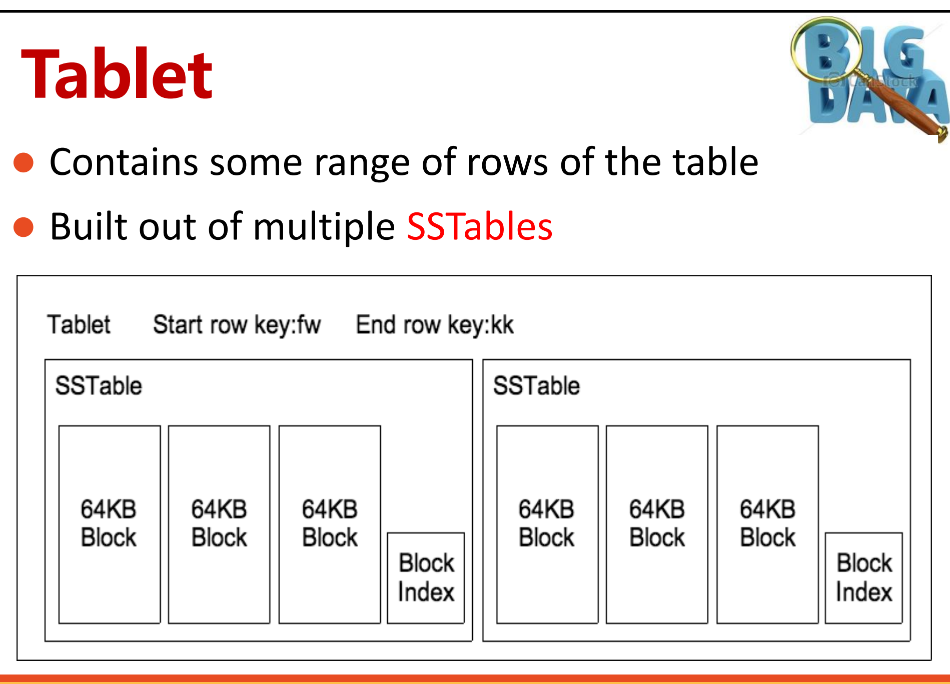


Tablet Storage

- Commit log on GFS – Redo log
  - buffered in tablet server's memory
- A set of locality groups
  - one locality group = a set of **SSTable** files on GFS
  - key = <row, column, timestamp>, value = cell content

SStable(Sorted String Table)

- SSTable: Sorted String Table
  - persistent, ordered, immutable map from keys to values.
    - keys and values are arbitrary byte strings.
    - SSTable: Immutable on-disk ordered map from string->string
    - string keys: <*row, column, timestamp*> triples
  - contains a sequence of blocks (typical size = 64KB), with a block index at the end of SSTable loaded at open time (next slide).
  - one disk seek per block read.
  - operations: lookup(key), iterate(key_range).
  - an SSTable can be mapped into memory.


Tablet
Contains some range of rows of the table
Built out of multiple SSTables
Tablet Start row key:fw End row key:kk
SSTable
64KB Block
64KB Block
64KB Block
Block Index
SSTable
64KB Block
64KB Block
64KB Block
Block Index
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 63

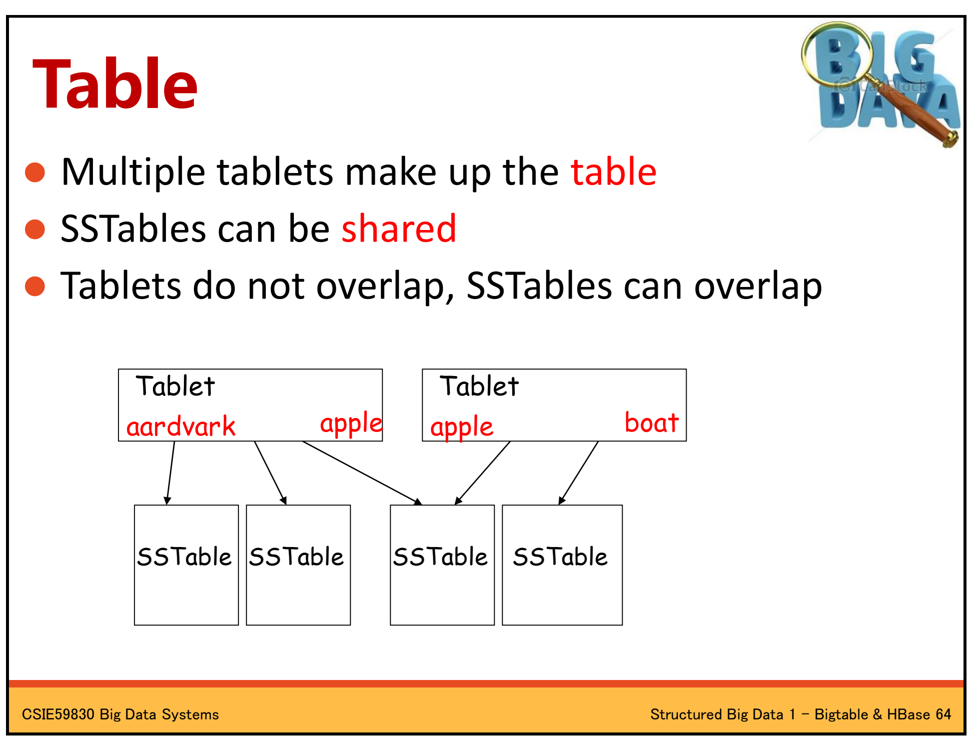
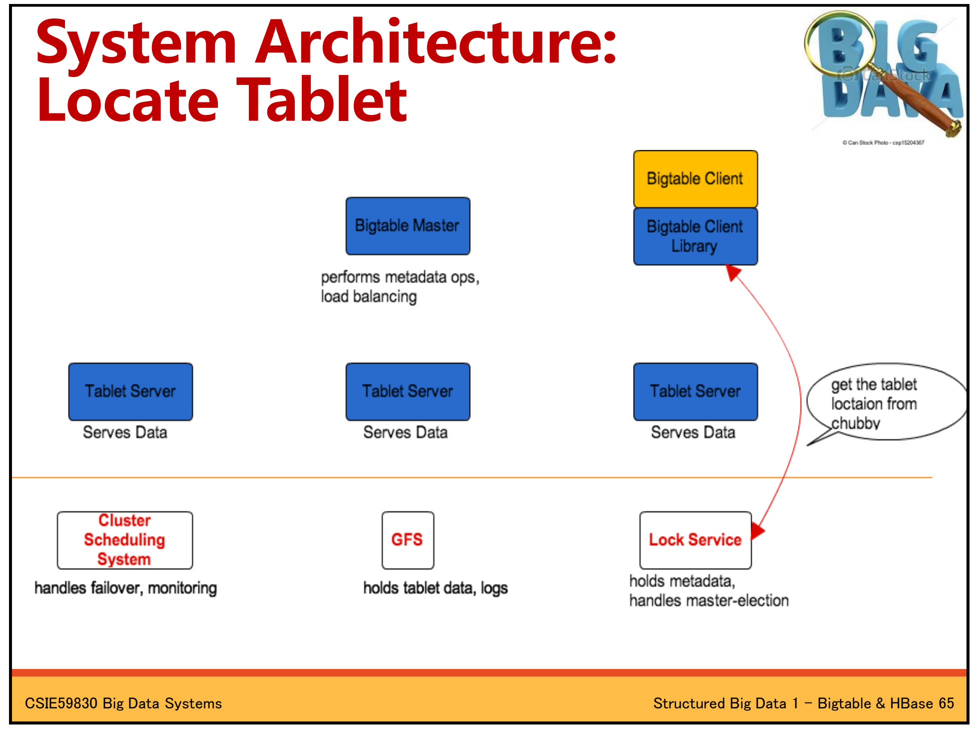
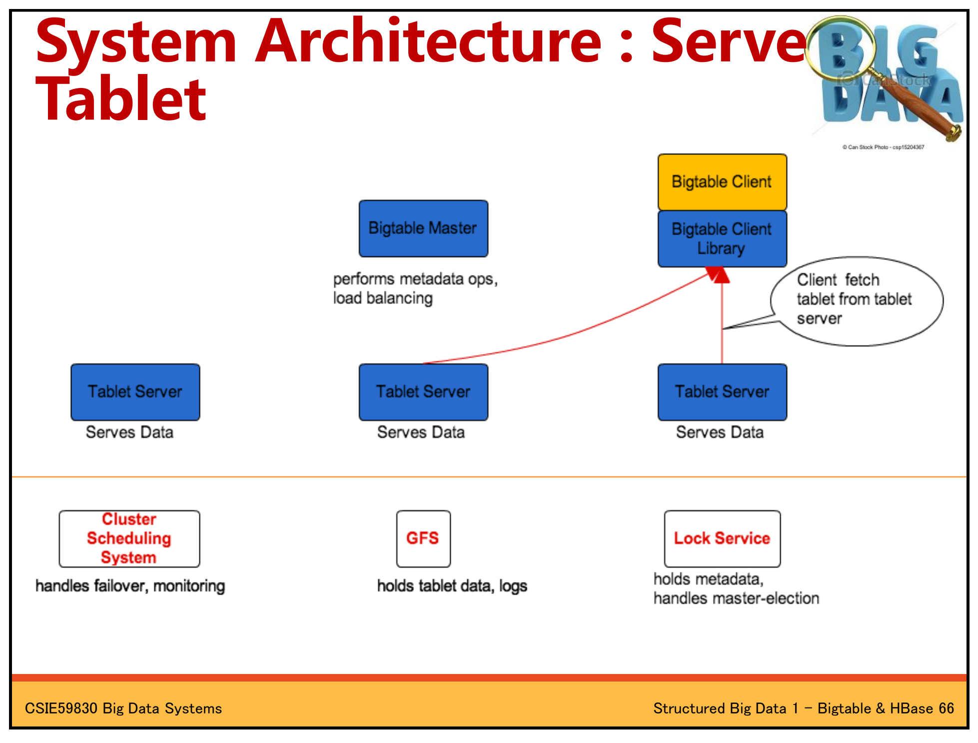
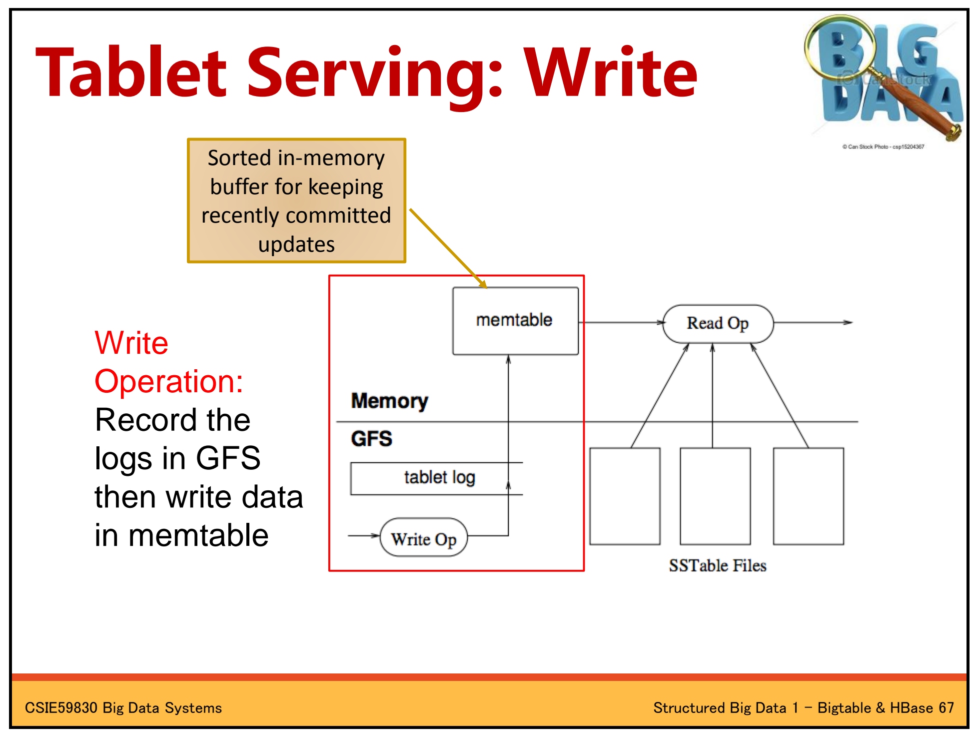
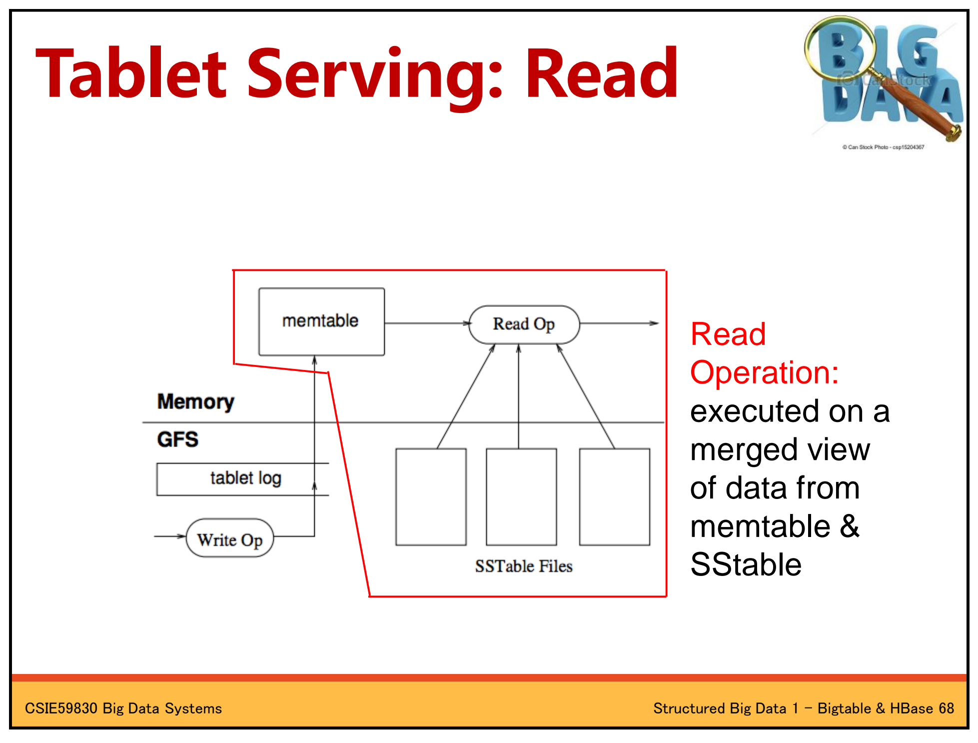


Table
Multiple tablets make up the table
SSTables can be shared
Tablets do not overlap, SSTables can overlap
Tablet
aardvark
apple
Tablet
apple
boat
SSTable
SSTable
SSTable
SSTable
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 64


System Architecture:
Locate Tablet
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
get the tablet loctaion from chubby
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 65


System Architecture : Serve Tablet
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Client fetch tablet from tablet server
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 66

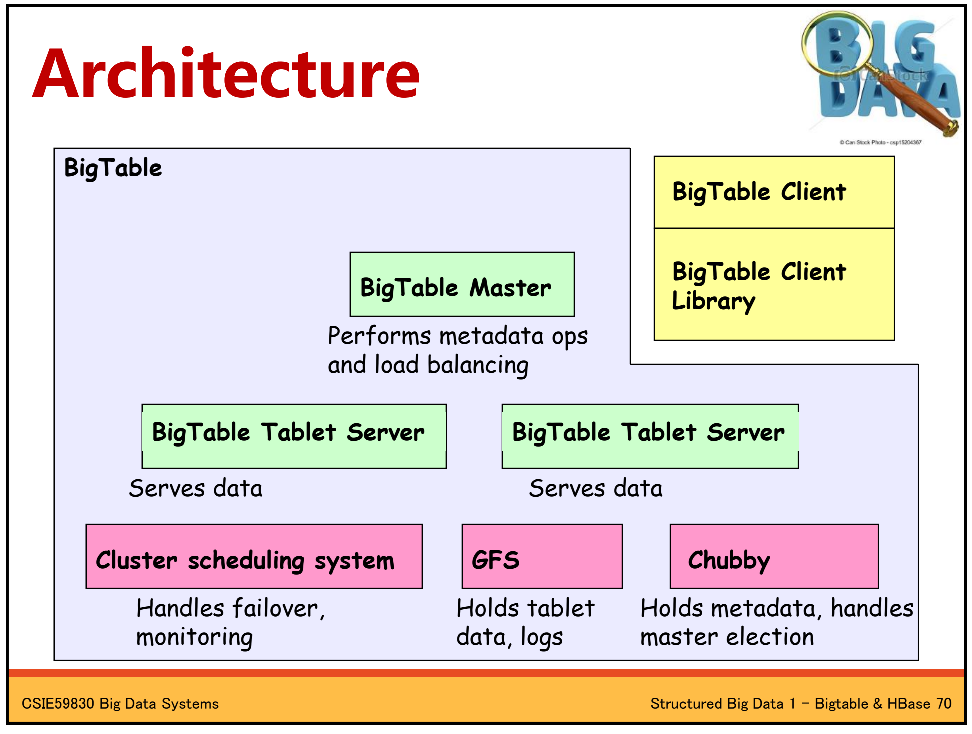

Tablet Serving: Write
Sorted in-memory buffer for keeping recently committed updates
Write Operation: Record the logs in GFS then write data in memtable
memtable
Read Op
Memory
GFS
tablet log
Write Op
SSTable Files
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 67


Tablet Serving: Read
memtable
Read Op
Memory
GFS
tablet log
Write Op
SSTable Files
Read Operation: executed on a merged view of data from memtable & SStable
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 68

Implementation: Three major components

- A library that is linked into every client
- One master server
  - Assigning tablets to tablet servers
  - Detecting the addition and deletion of tablet servers
  - Balancing tablet-server load
  - Garbage collection of files in GFS
- Many tablet servers
  - Tablet servers manage tablets
  - Tablet server splits tablets that get too big
- Client communicates directly with tablet server for reads/writes.

Architecture



Tablet Server

- When a tablet server starts, it creates and acquires exclusive lock on a uniquely-named file in a specific Chubby directory
  - Call this servers directory
- A tablet server stops serving its tablets if it loses its exclusive lock
  - This may happen if there is a network connection failure that causes the tablet server to lose its Chubby session

Tablet Server

- A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists
- If the file no longer exists then the tablet server will never be able to serve again
  - Kills itself
  - At some point it can restart; it goes to a pool of unassigned tablet servers

Master Startup Operation

- Upon start up the master needs to discover the current tablet assignment.
  - Grabs unique master lock in Chubby
    - Prevents concurrent master instantiations
  - Scans servers directory in Chubby for live servers
  - Communicates with every live tablet server
    - Discover all tablets
  - Scans METADATA table to learn the set of tablets
    - Unassigned tablets are marked for assignment

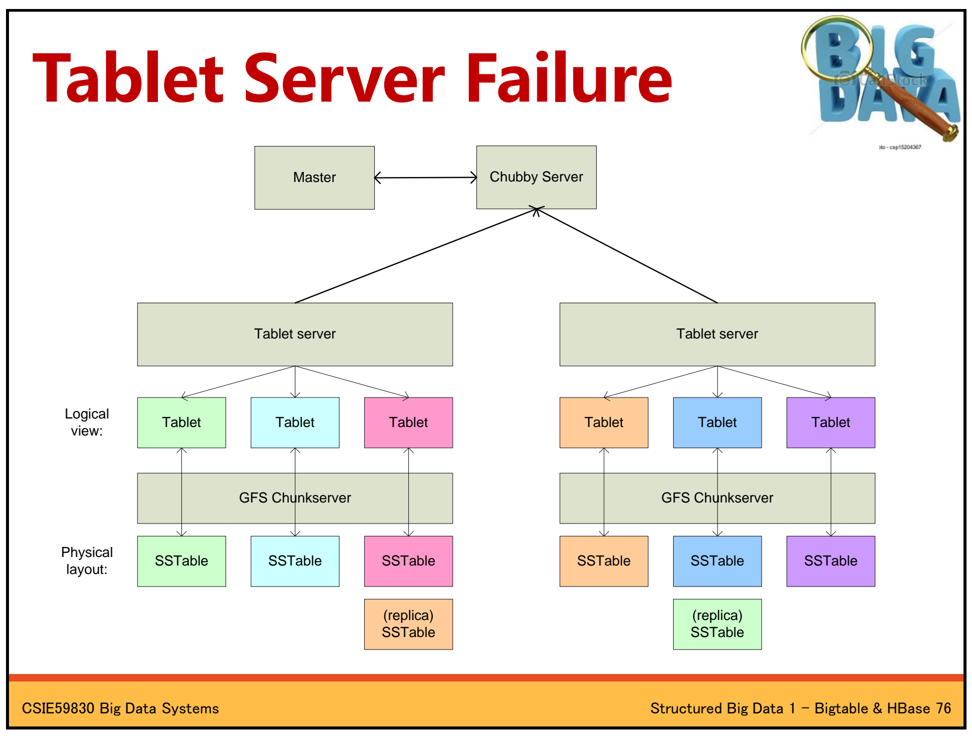
Master Operation

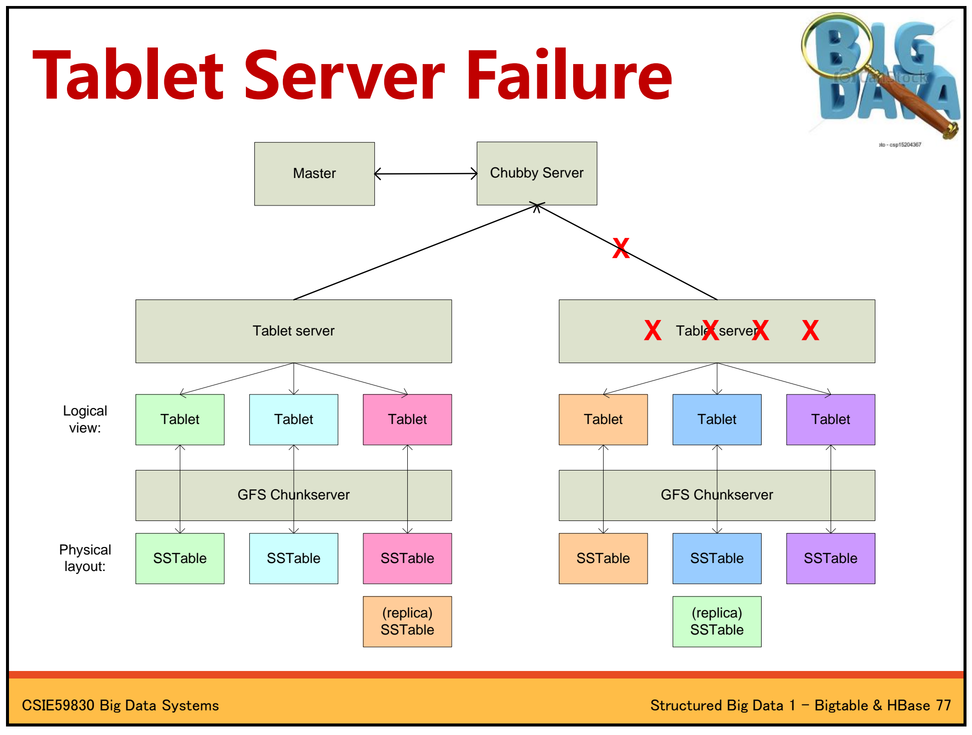
- Detect tablet server failures/resumption
- Master periodically asks each tablet server for the status of its lock

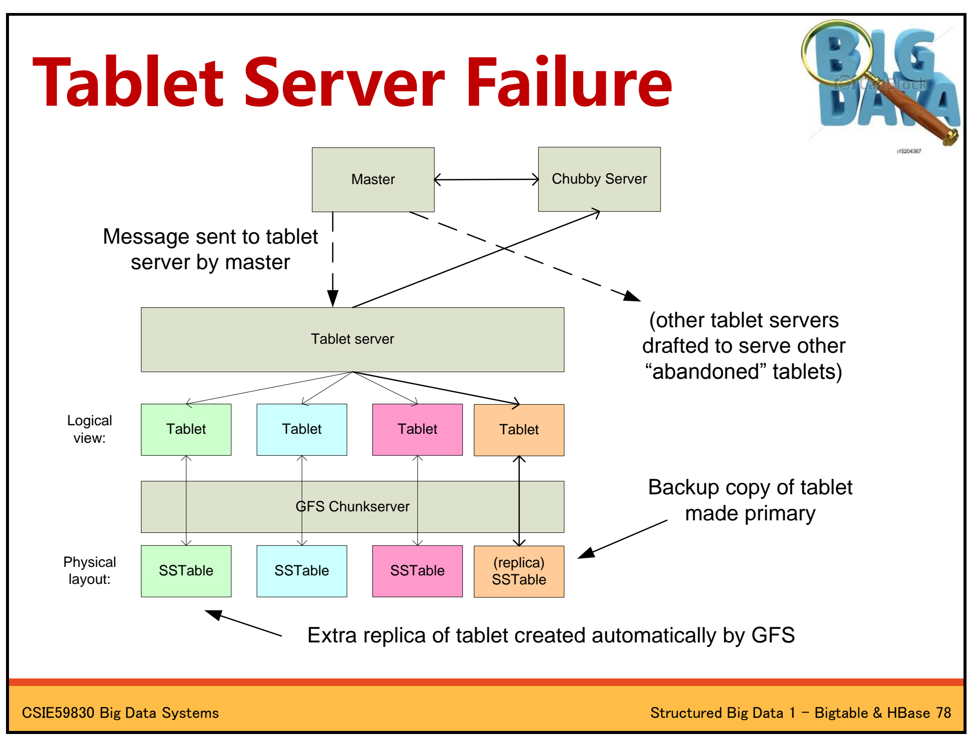
Master Operation

- Tablet server lost its lock or master cannot contact tablet server:
  - Master attempts to acquire exclusive lock on the server's file in the servers directory
  - If master acquires the lock then the tablets assigned to the tablet server are assigned to others
    - Master deletes the server's file in the servers directory
  - Assignment of tablets should be balanced
- If master loses its Chubby session then it kills itself
  - An election can take place to find a new master

Tablet Server Failure

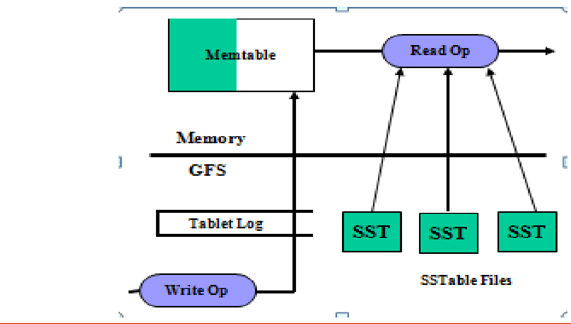



Tablet Server Failure
Master
Chubby Server
X
Tablet server
X Tablet server X X X
Logical view:
Tablet
Tablet
Tablet
Tablet
Tablet
Tablet
GFS Chunkserver
GFS Chunkserver
Physical layout:
SSTable
SSTable
SSTable
SSTable
SSTable
SSTable
(replica) SSTable
(replica) SSTable
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 77


Tablet Server Failure
Master
Chubby Server
Message sent to tablet server by master
(other tablet servers drafted to serve other "abandoned" tablets)
Tablet server
Logical view:
Tablet
Tablet
Tablet
Tablet
GFS Chunkserver
Backup copy of tablet made primary
Physical layout:
SSTable
SSTable
SSTable
(replica) SSTable
Extra replica of tablet created automatically by GFS
CSIE59830 Big Data Systems
Structured Big Data 1 – Bigtable & HBase 78

Tablet Serving

- Commit log stores the updates that are made to the data
- Recent updates are stored in memtable
- Older updates are stored in SStable files



Tablet Serving

- Recovery process
- Reads/Writes that arrive at tablet server
  - Is the request well-formed?
  - Authorization: Chubby holds the permission file
  - If a mutation occurs it is wrote to commit log and finally a group commit is used

Tablet Serving

- Tablet recovery process
  - Read metadata containing SSTables and redo points
  - Redo points are pointers into any commit logs
  - Apply redo points

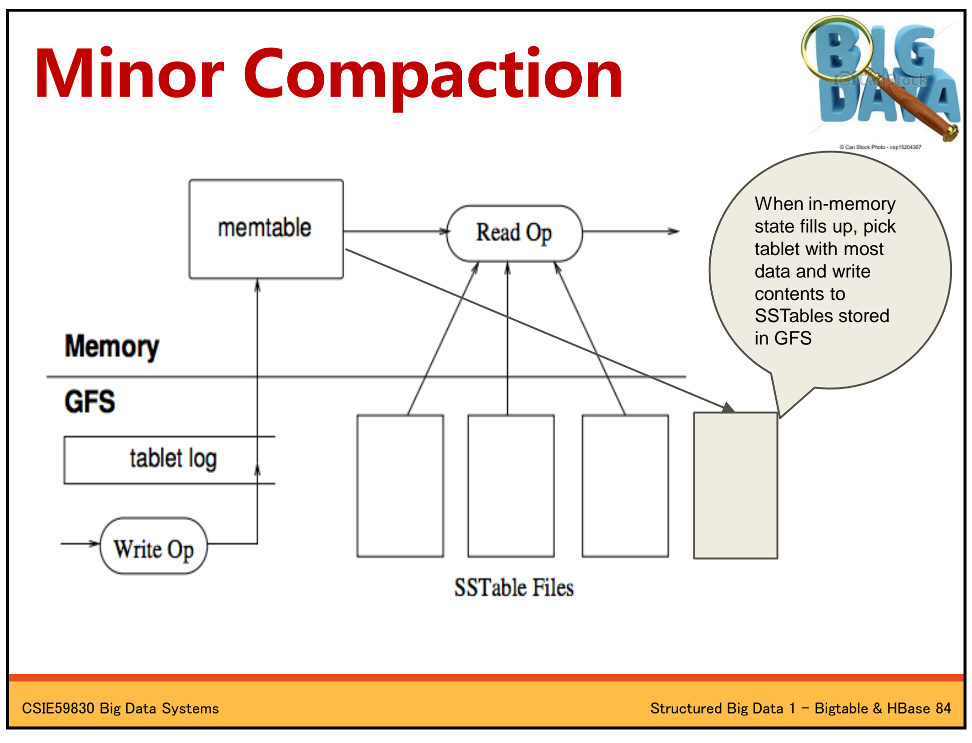
Compactions

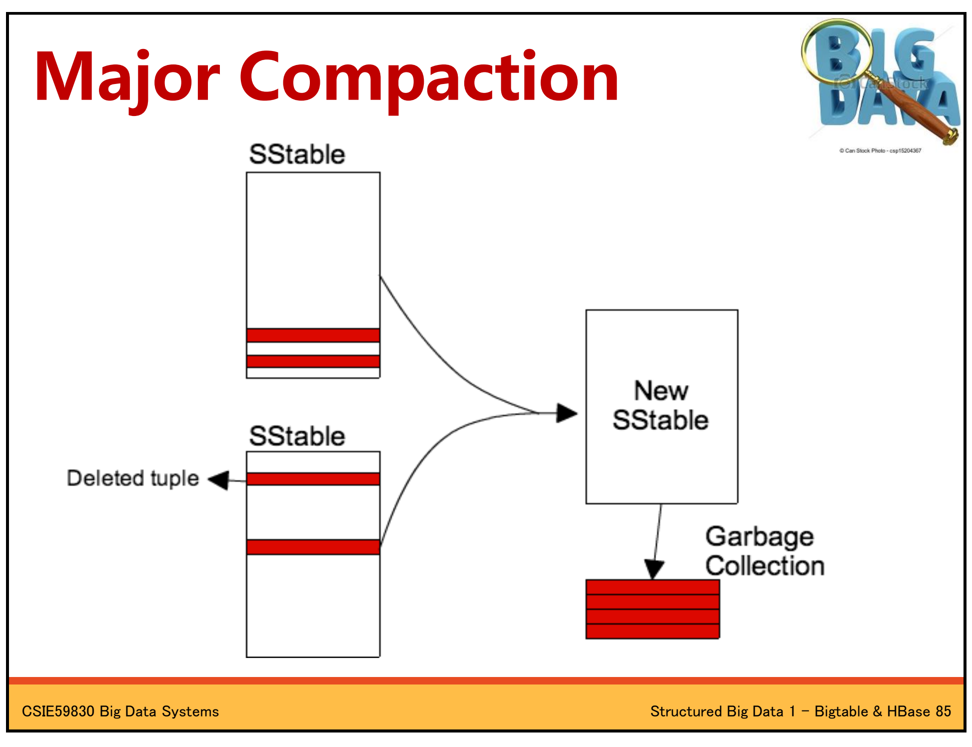
- As writes execute, size of memtable increases.
- Once memtable reaches a threshold:
  - Memtable is frozen,
  - A new memtable is created,
  - Frozen metable is converted to an SSTable and written to GFS.
- This **minor compaction** – convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic and recovery time on restart

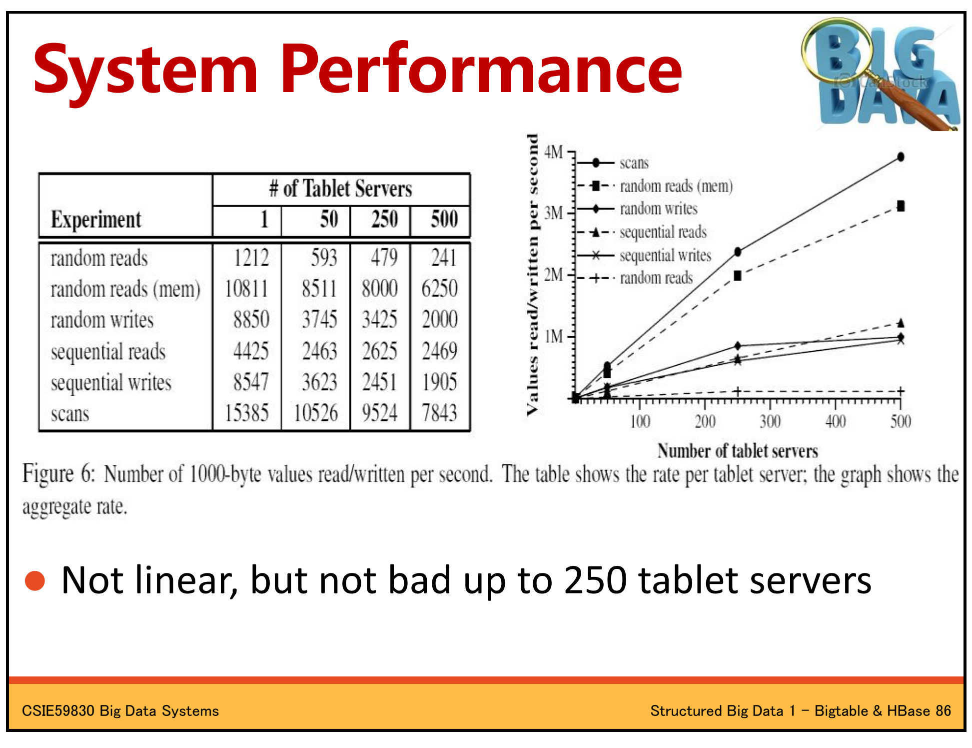
Compactions

- **Merging compaction** (in the background)
  - Read a few SSTables and memtable to produce one SSTable. (Input SSTables and memtable are discareded.)
  - Reduce number of SSTables
  - Good place to apply policy “keep only N versions”
- **Major compaction**
  - Periodically compact all SSTables for tablet into a new base SSTable on GFS
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

Minor Compaction







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.

- Not linear, but not bad up to 250 tablet servers

Performance Observation

- Random reads slow because tablet server channel to GFS saturated
- Random reads (mem) is fast because only memtable involved
- Random & sequential writes > sequential reads because only log and memtable involved
- Sequential read > random read because of block caching
- Scans even faster because tablet server can return more data per RPC

Refinements

- **Locality groups**
  - Clients can group multiple column families together into a *locality group.*
- **Compression**
  - Compression applied to each SSTable block separately
  - Uses *Bentley and McIlroy's* scheme and *fast compression* algorithm
- **Caching for read performance**
  - Uses Scan Cache and Block Cache
- **Bloom filters**
  - Reduce the number of disk accesses

Refinements

- Commit-log implementation
  - Suppose one log per tablet rather than one log per tablet server
- Exploiting SSTable immutability
  - No need to synchronize accesses to file system when reading SSTables
  - Concurrency control over rows efficient
  - Deletes work like garbage collection on removing obsolete SSTables
  - Enables quick tablet split: parent SSTables used by children

CAP Revisit

- Consistency
  - Everybody see the same result of an operation
- Availability
  - No matter an operation succeeds or fails, a result must be returned -- the system must respond
- Partition Tolerance
  - The system must work still despite of message loss or node failure -- communication within cluster

CAP Theorem

- The CAP theorem: in distributed system, consistency, availability & partition tolerance can't be fulfilled together.
- Proposed by E. Brewer of UCB as a conjecture
- Proved by Seth Gilbert and Nancy Lynch of MIT

CAP on Bigtable

- Bigtable is a distributed database
- Something must be sacrificed
  - **P**artition tolerance is required: things will fail
  - **C**onsistency is fulfilled: row atomicity
  - Availability not fulfilled: what if Chubby fails?
- Consistency is more important for their applications than availability
- Other systems may have different goals

NoSQL Databases

- NoSQL stands for "not only SQL"
- The type of systems for structured big data with SQL-like capabilities
- Arise in the big data era
- Must trade off between C, A and P.