Structured Big Data 1: 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

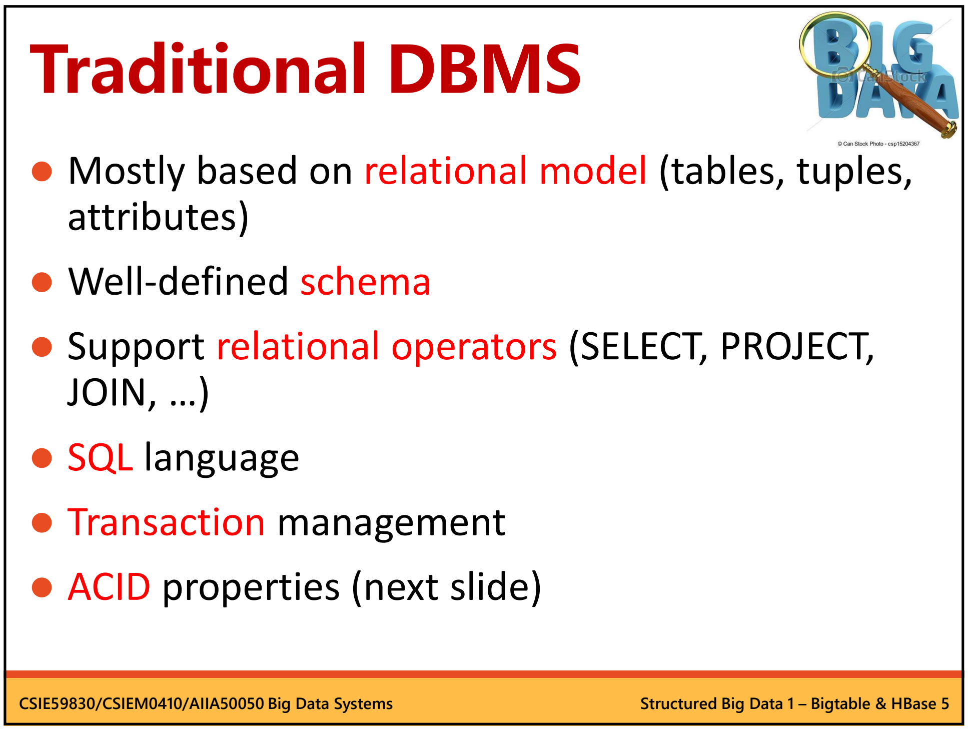
- Problems of big data processing with Hadoop MapReduce
- Structured big data processing
- Traditional RDBMS
- ACID vs BASE
- Distributed DB
- Google Bigtable
- Apache Hbase
- CAP theorem

Problems of MR 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.

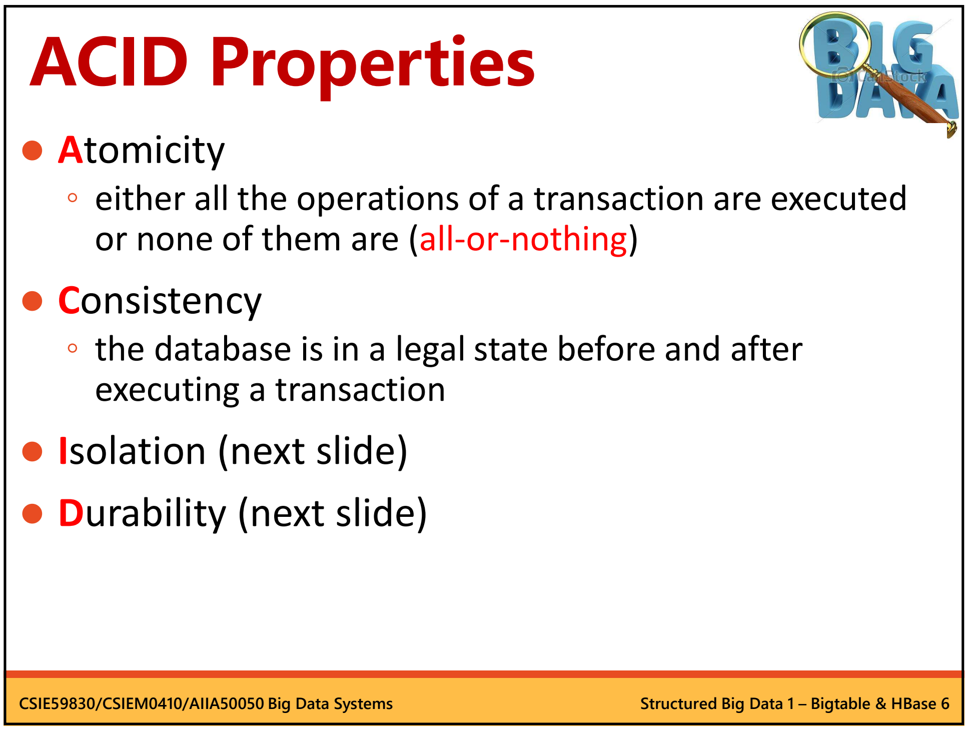
Structured Big Data Processing

- 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.
- Continue with NoSQL, NewSQL and Distributed SQL systems. (next lecture)



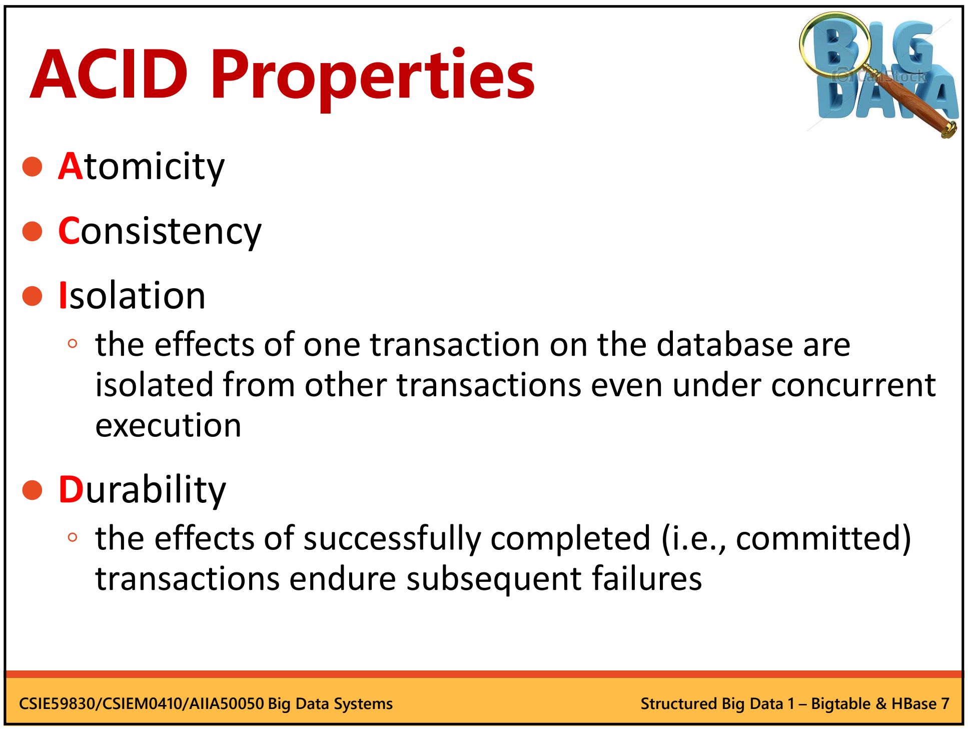
Traditional DBMS

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



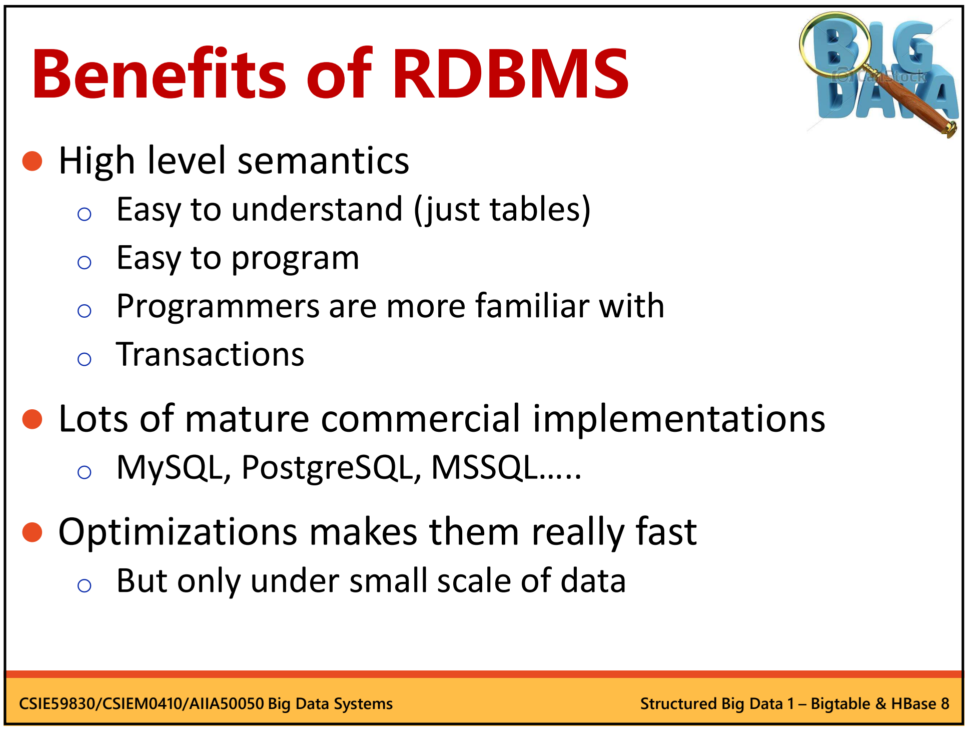
ACID Properties

- **A**tomicity
  - either all the operations of a transaction are executed or none of them are (all-or-nothing)
- **C**onsistency
  - the database is in a legal state before and after executing a transaction
- **I**solation (next slide)
- **D**urability (next slide)



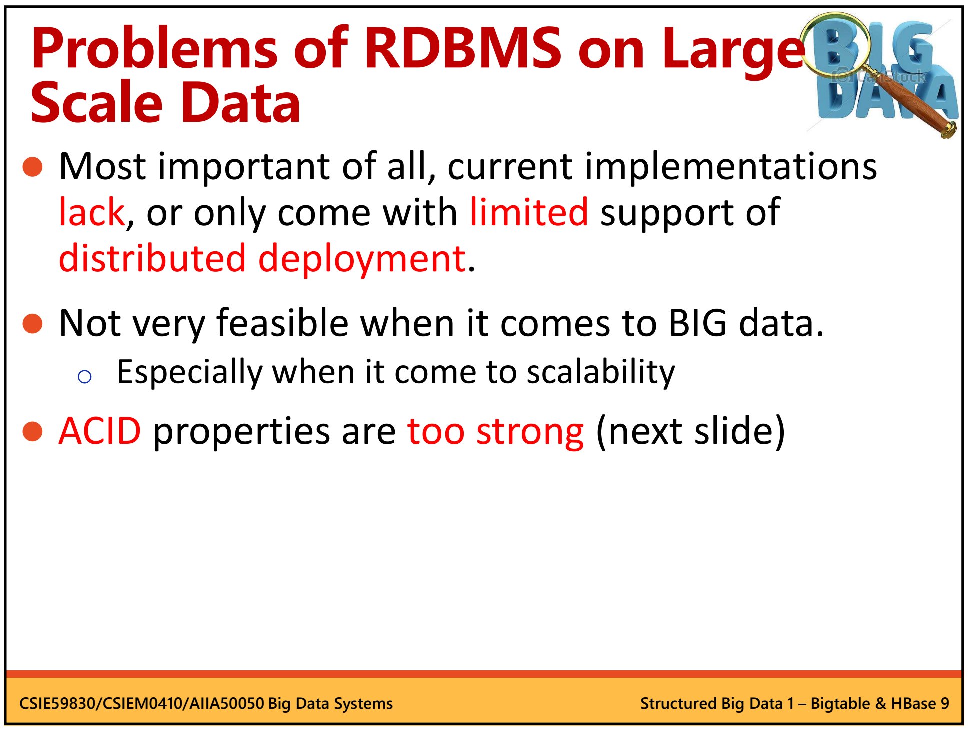
ACID Properties

- 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



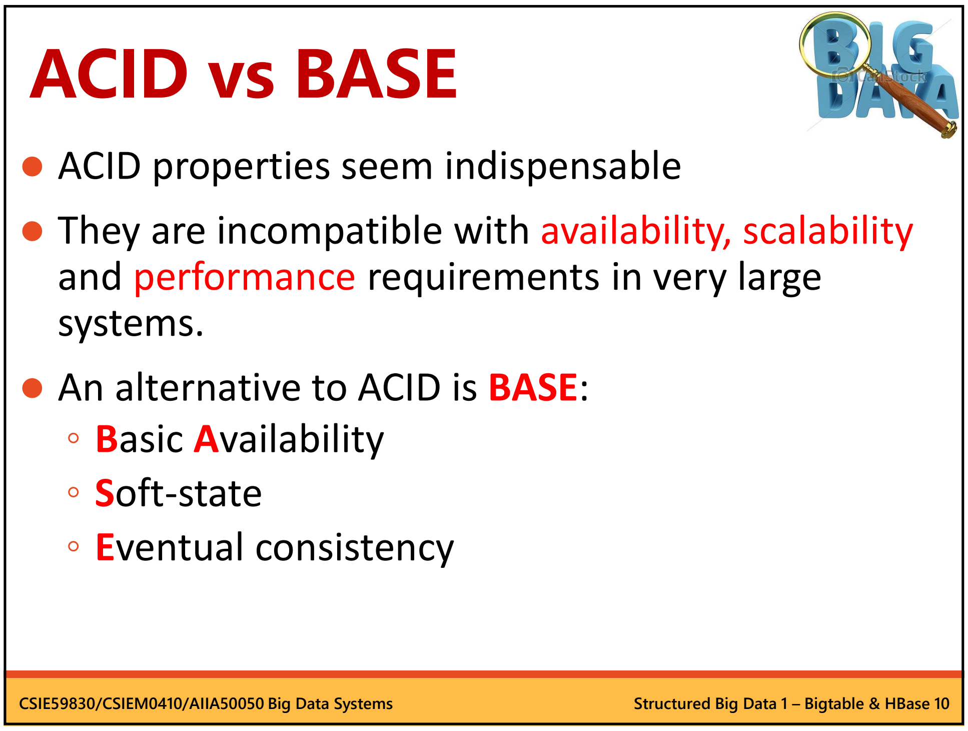
Benefits of RDBMS

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



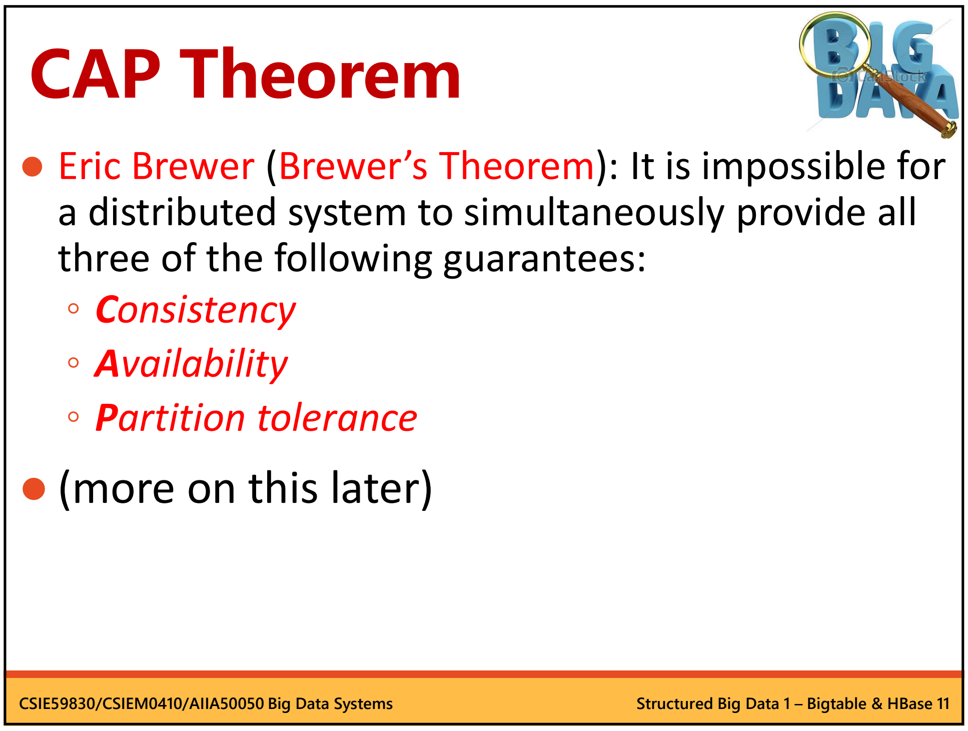
Problems of RDBMS on Large Scale Data

- 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)



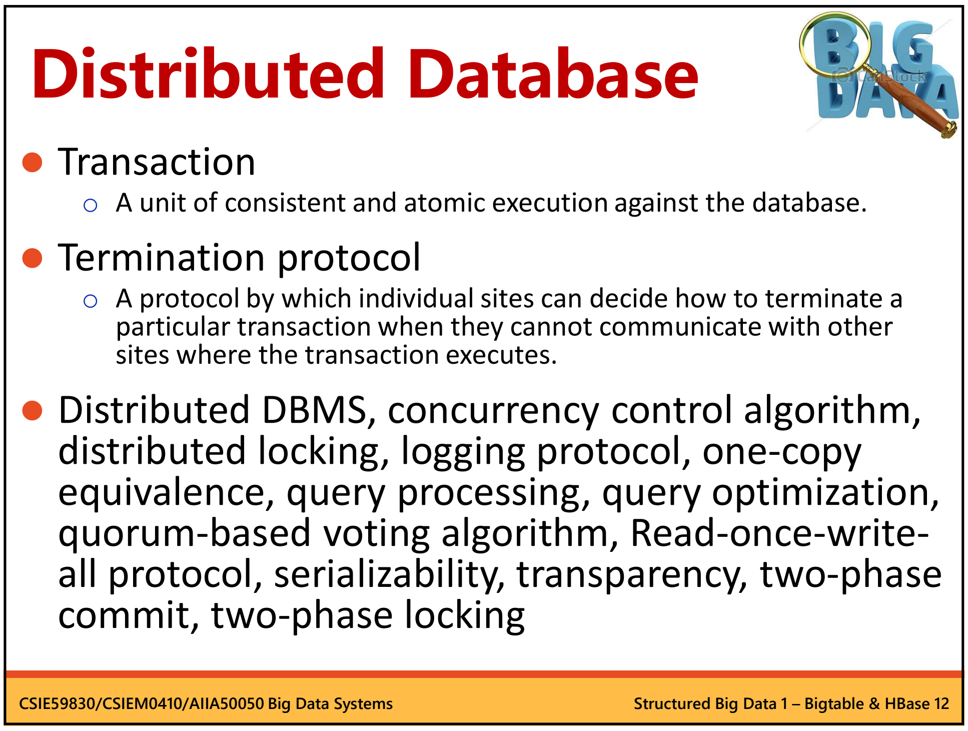
ACID vs BASE

- 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



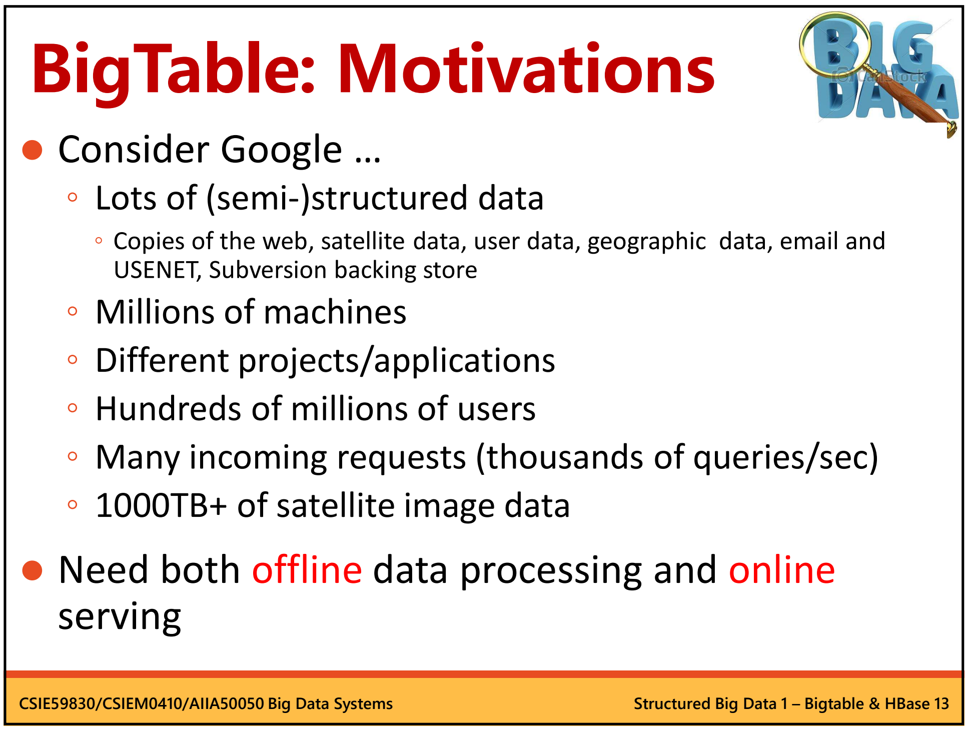
CAP Theorem

- 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)



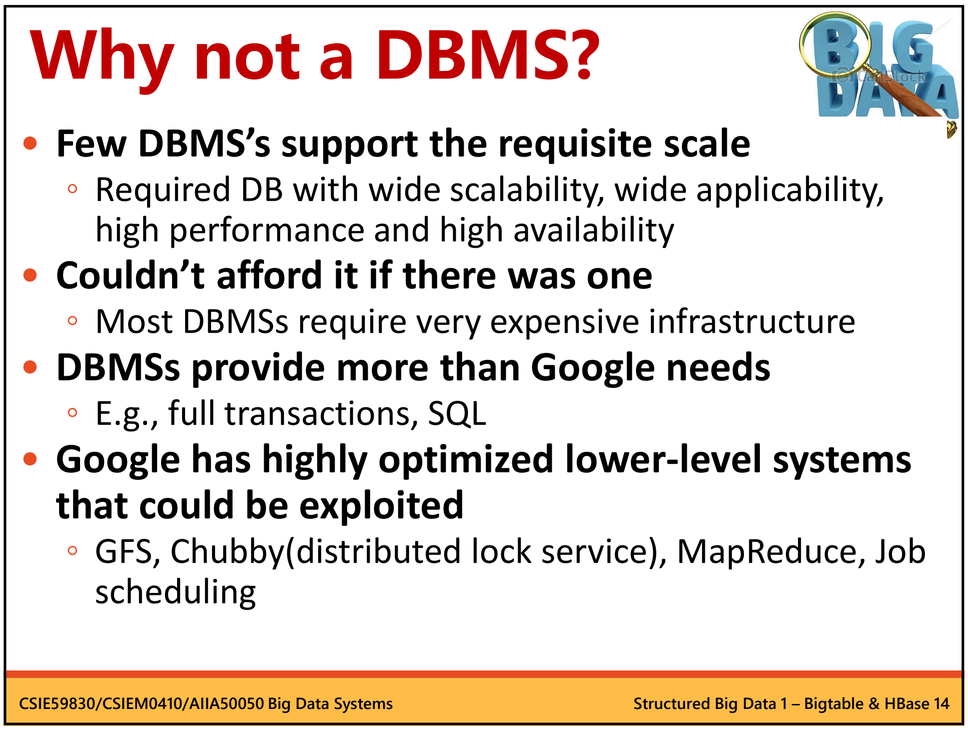
Distributed Database

- 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



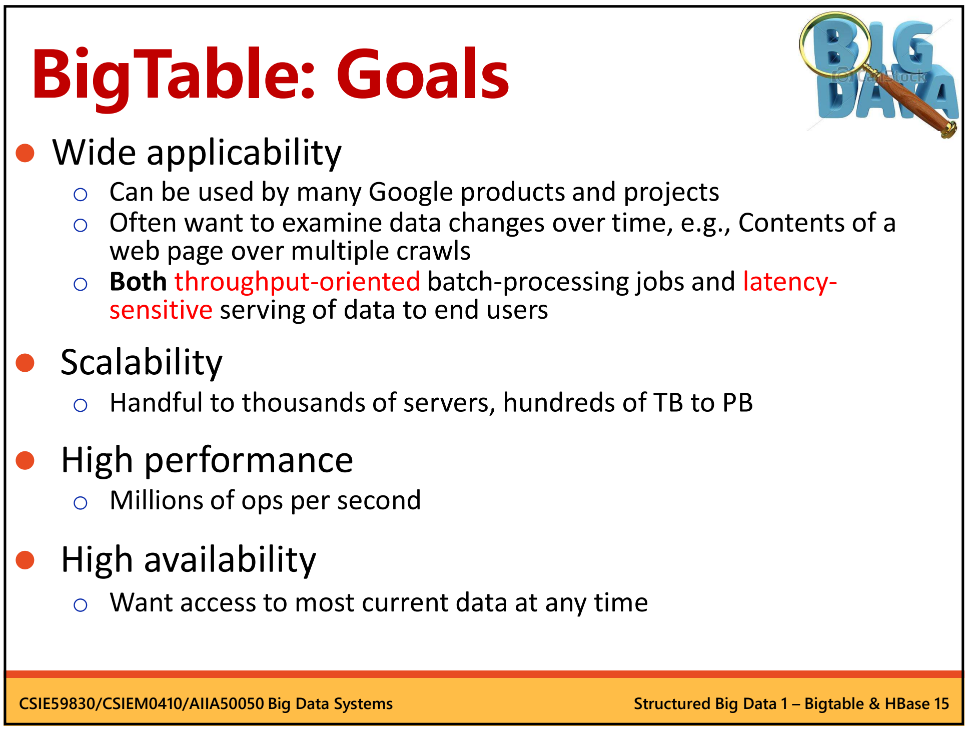
BigTable: Motivations

- 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 queries/sec)
  - 1000TB+ of satellite image data
- Need both offline data processing and online serving



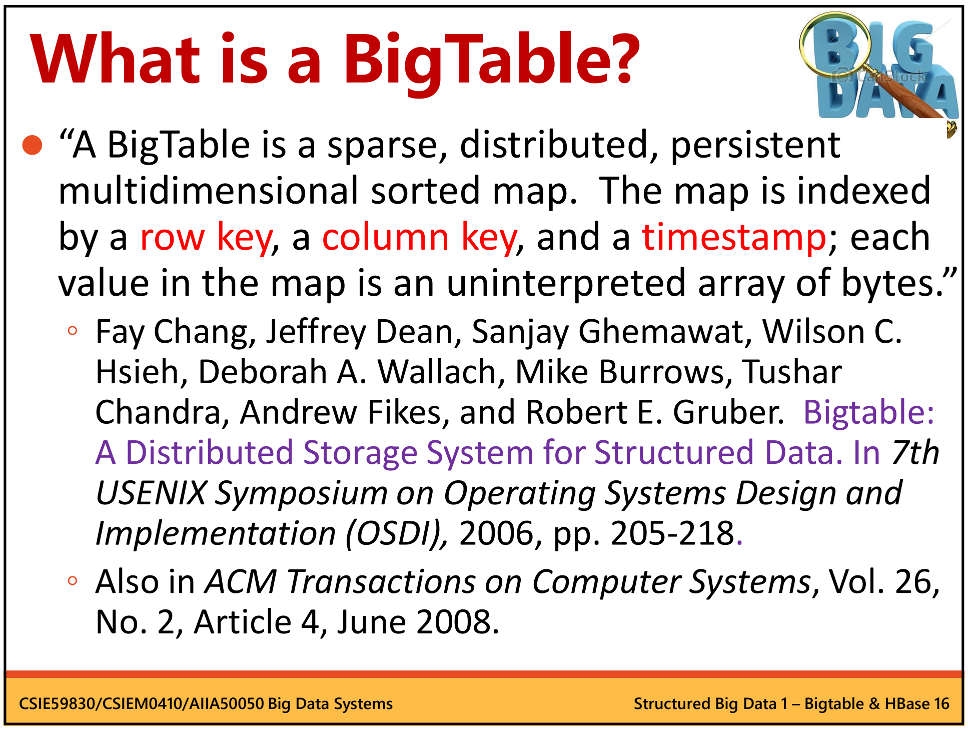
Why not a DBMS?

- **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



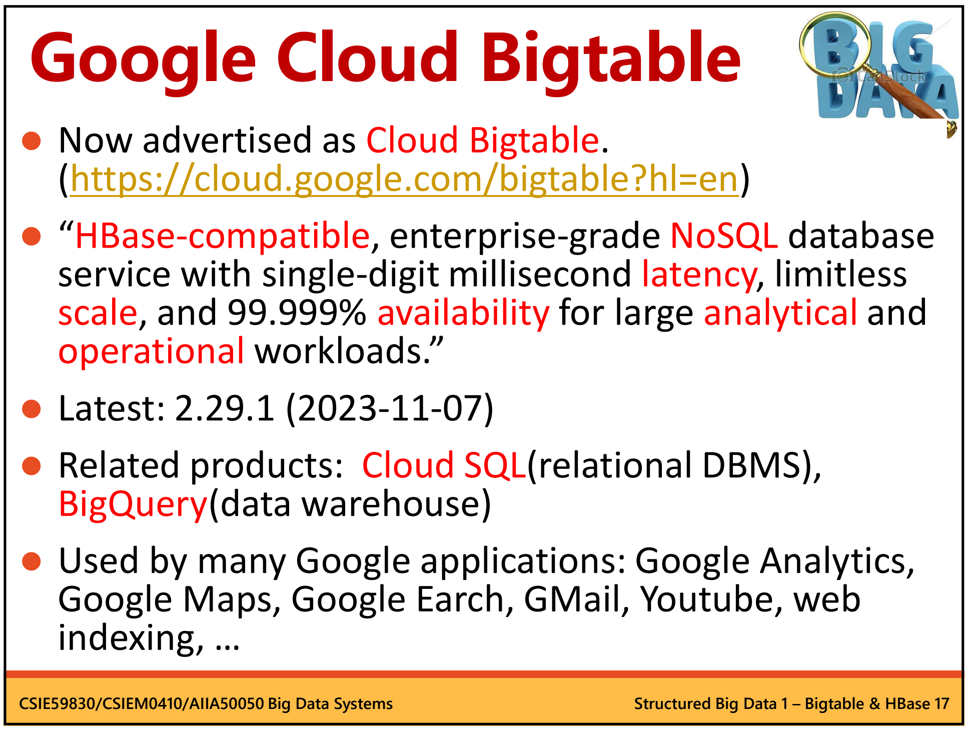
BigTable: Goals

- 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



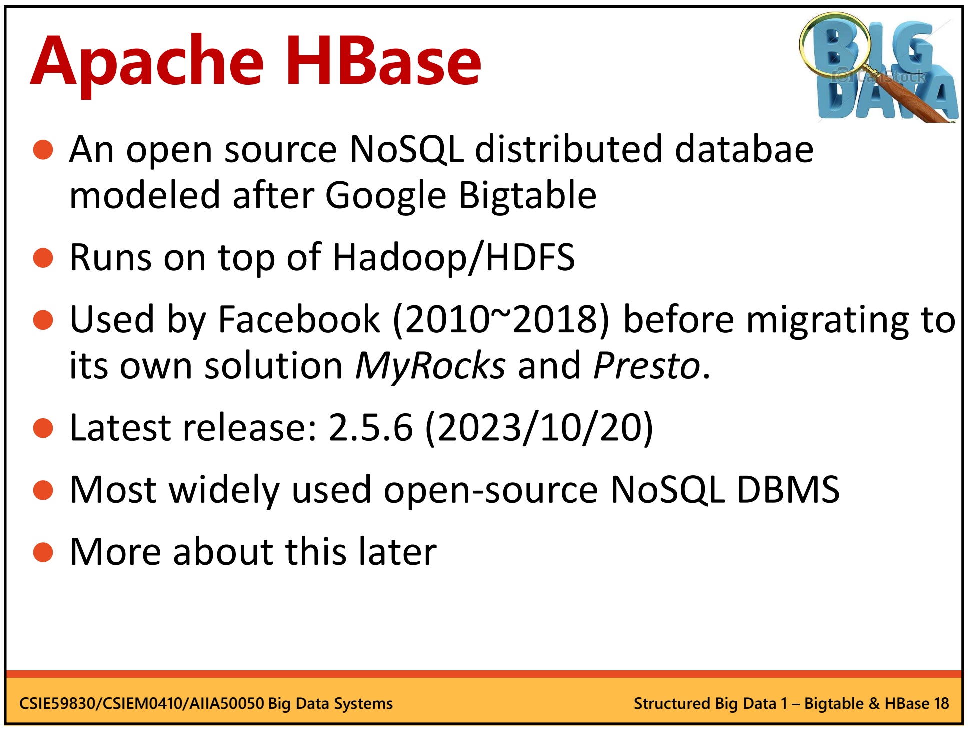
What is a BigTable?

- "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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205-218.
  - Also in *ACM Transactions on Computer Systems*, Vol. 26, No. 2, Article 4, June 2008.



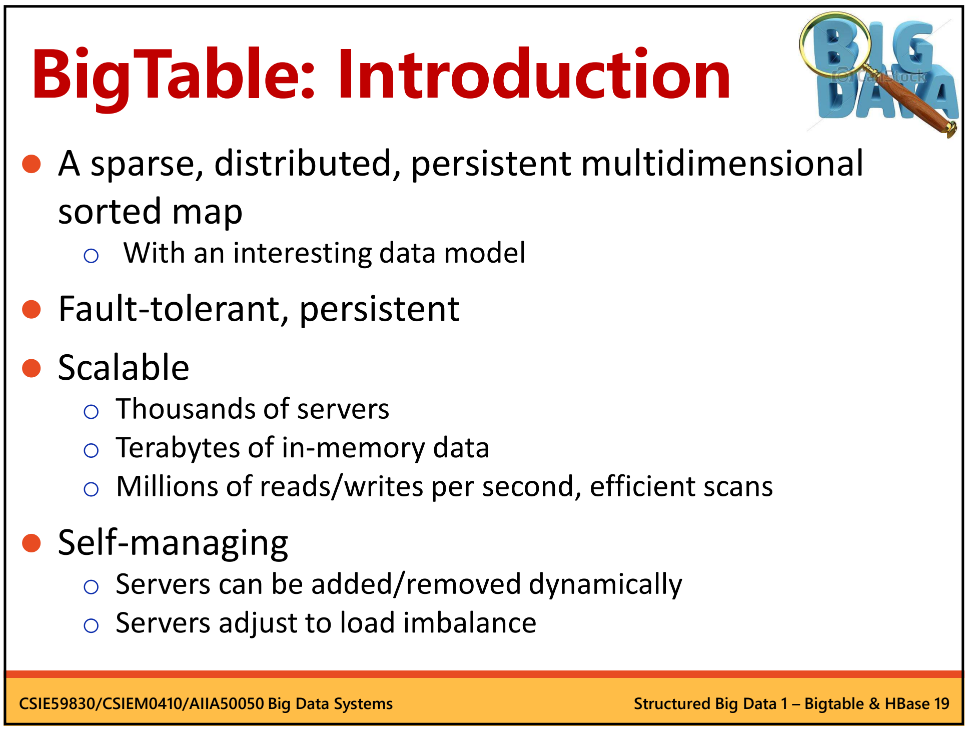
Google Cloud Bigtable

- Now advertised as Cloud Bigtable. (https://cloud.google.com/bigtable?hl=en)
- "HBase-compatible, enterprise-grade NoSQL database service with single-digit millisecond latency, limitless scale, and 99.999% availability for large analytical and operational workloads."
- Latest: 2.29.1 (2023-11-07)
- Related products: Cloud SQL(relational DBMS), BigQuery(data warehouse)
- Used by many Google applications: Google Analytics, Google Maps, Google Earch, GMail, Youtube, web indexing, …



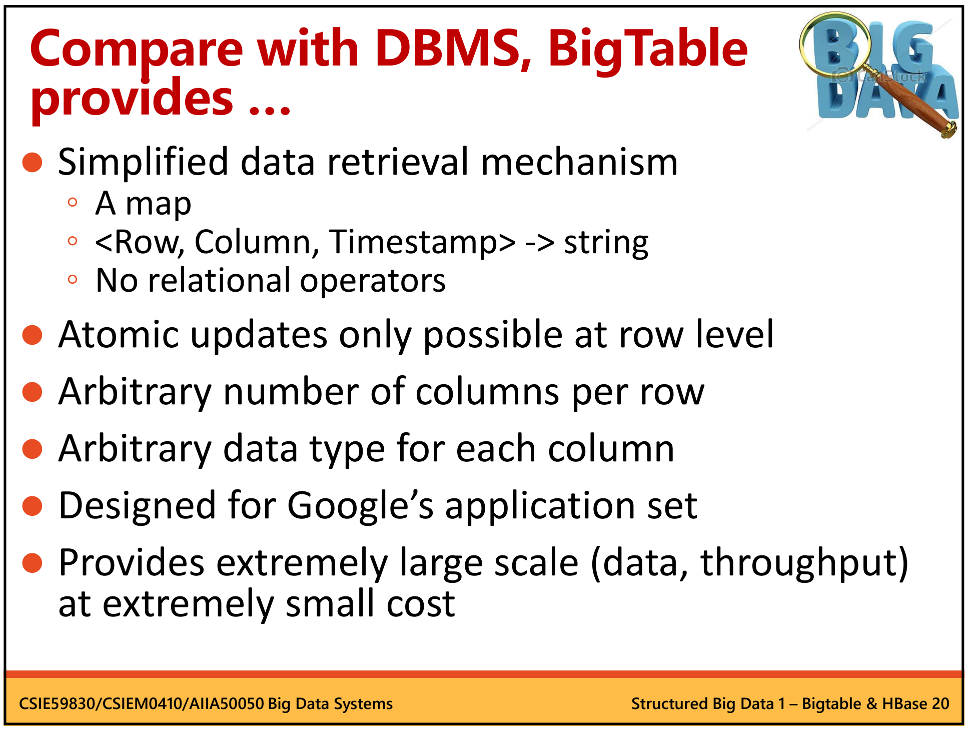
Apache HBase

- An open source NoSQL distributed databae modeled after Google Bigtable
- Runs on top of Hadoop/HDFS
- Used by Facebook (2010~2018) before migrating to its own solution *MyRocks* and *Presto*.
- Latest release: 2.5.6 (2023/10/20)
- Most widely used open-source NoSQL DBMS
- More about this later



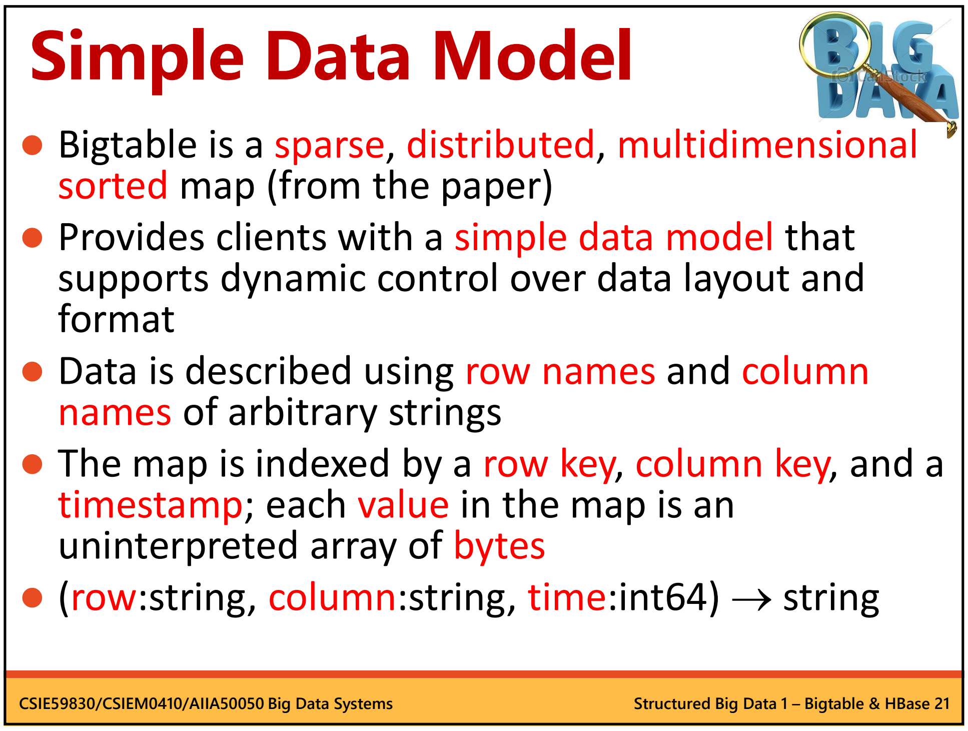
BigTable: Introduction

- 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



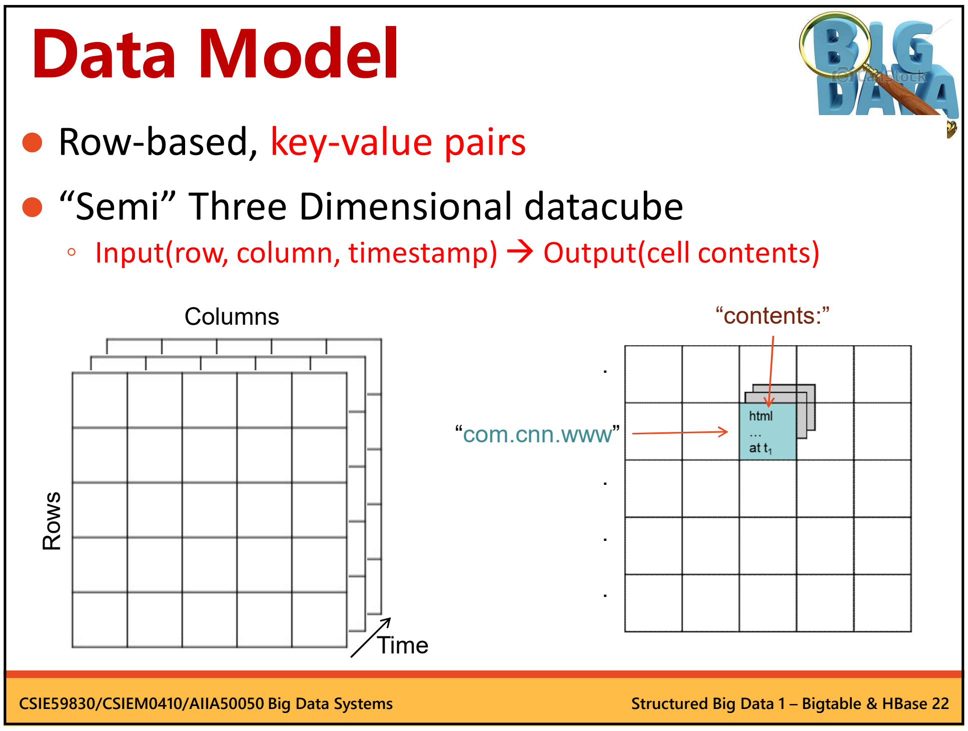
Compare with DBMS, BigTable provides ...

- 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



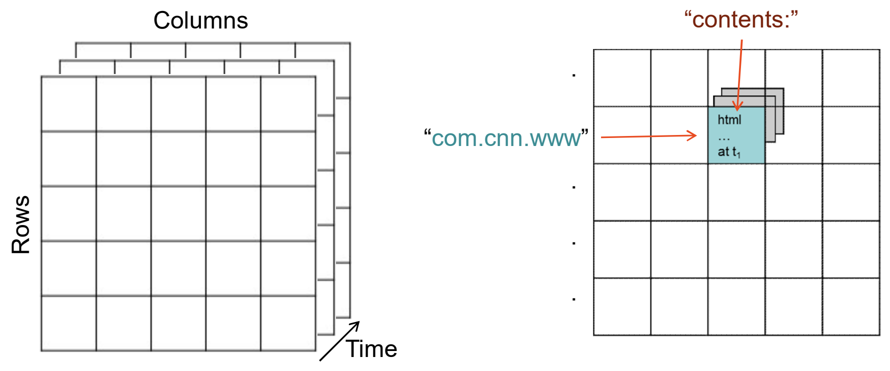
Simple Data Model

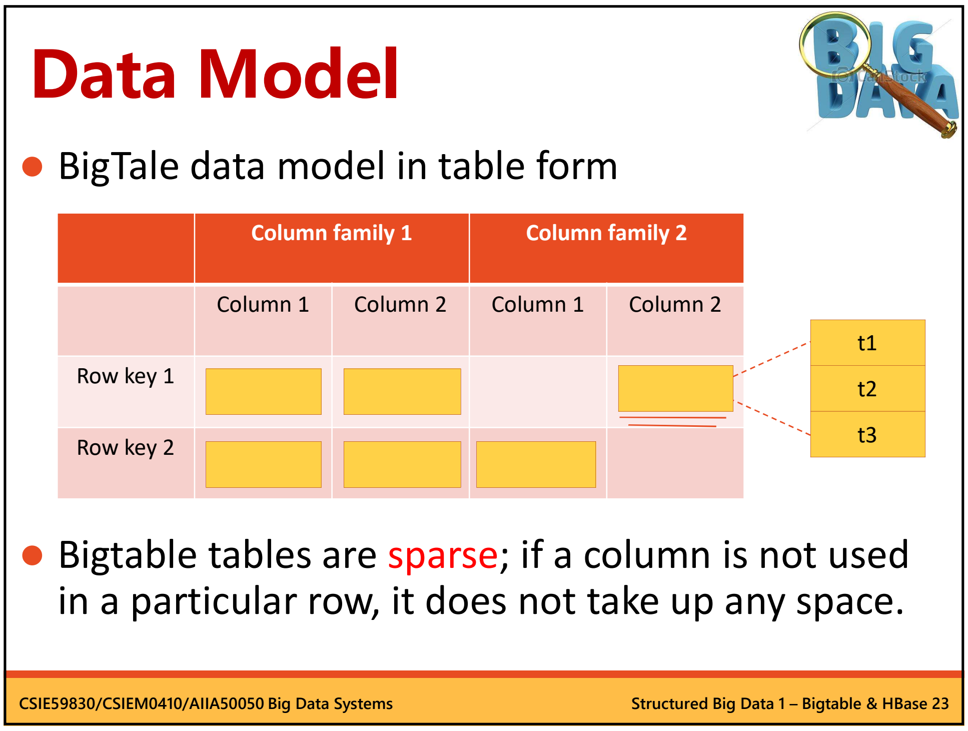
- Bigtable is a sparse, distributed, multidimensional sorted map (from the paper)
- Provides clients with a simple data model that supports dynamic control over data layout and format
- Data is described using row names and column names of arbitrary strings
- The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes
- (row:string, column:string, time:int64) → string



Data Model

- Row-based, key-value pairs
- “Semi” Three Dimensional datacube
  - Input(row, column, timestamp) → Output(cell contents)





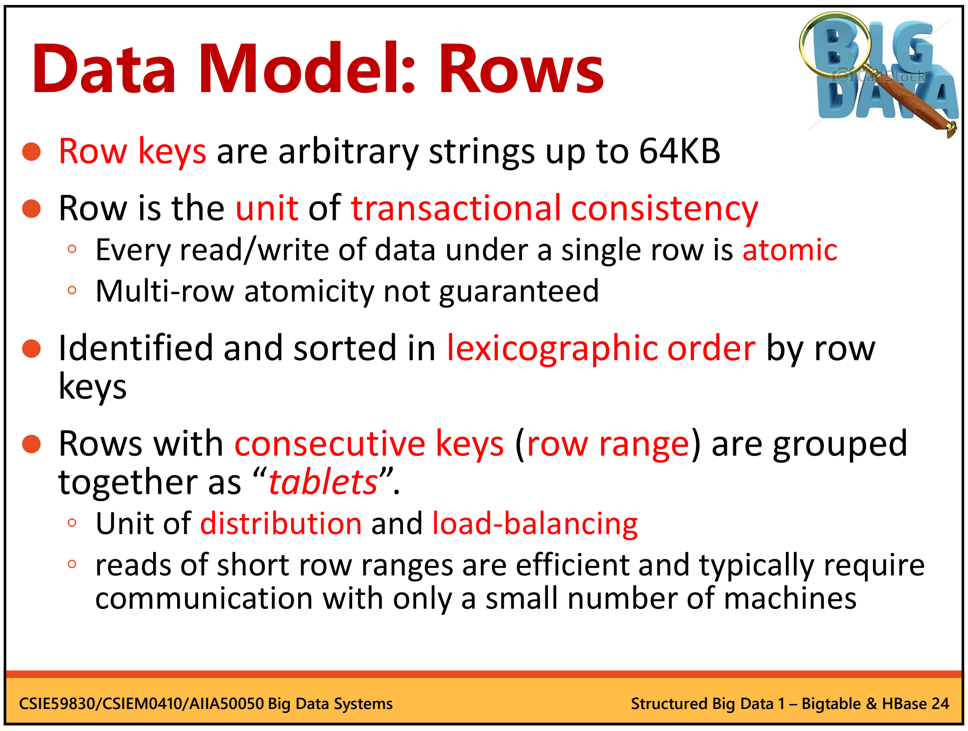
Data Model

- BigTale data model in table form

	Column family 1		Column family 2	
	Column 1	Column 2	Column 1	Column 2
Row key 1	🟨	🟨		🟨
Row key 2	🟨	🟨	🟨	

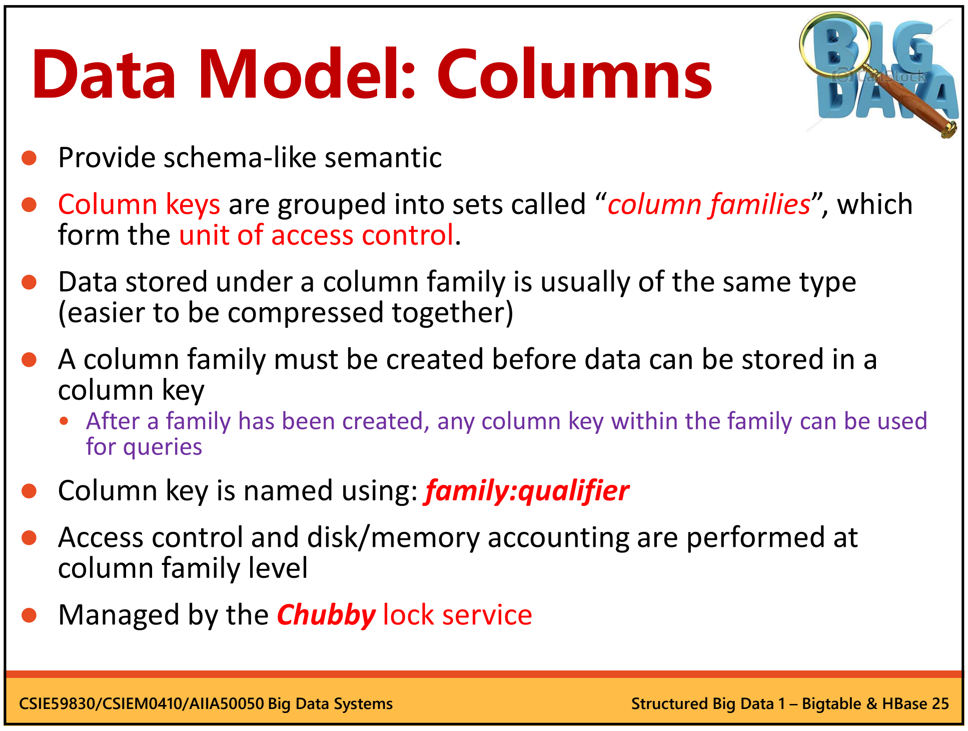
t1
t2
t3

- Bigtable tables are sparse; if a column is not used in a particular row, it does not take up any space.



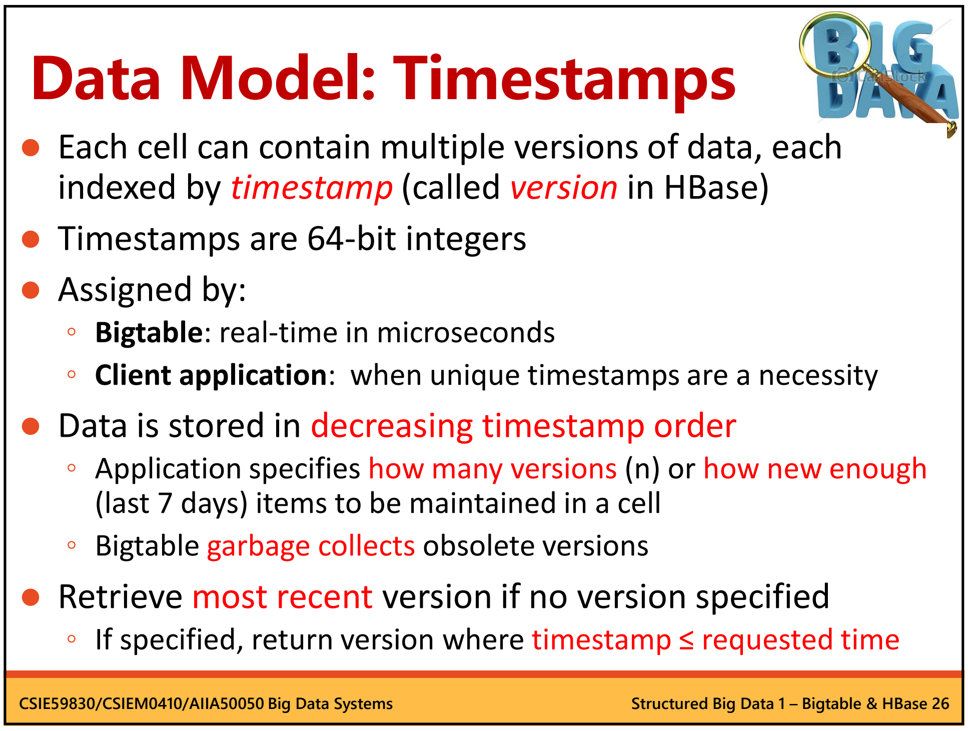
Data Model: Rows

- Row keys are arbitrary strings up to 64KB
- Row is the unit of transactional consistency
  - Every read/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
  - 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
- Retrieve most recent version if no version specified
  - If specified, return version where timestamp ≤ requested time

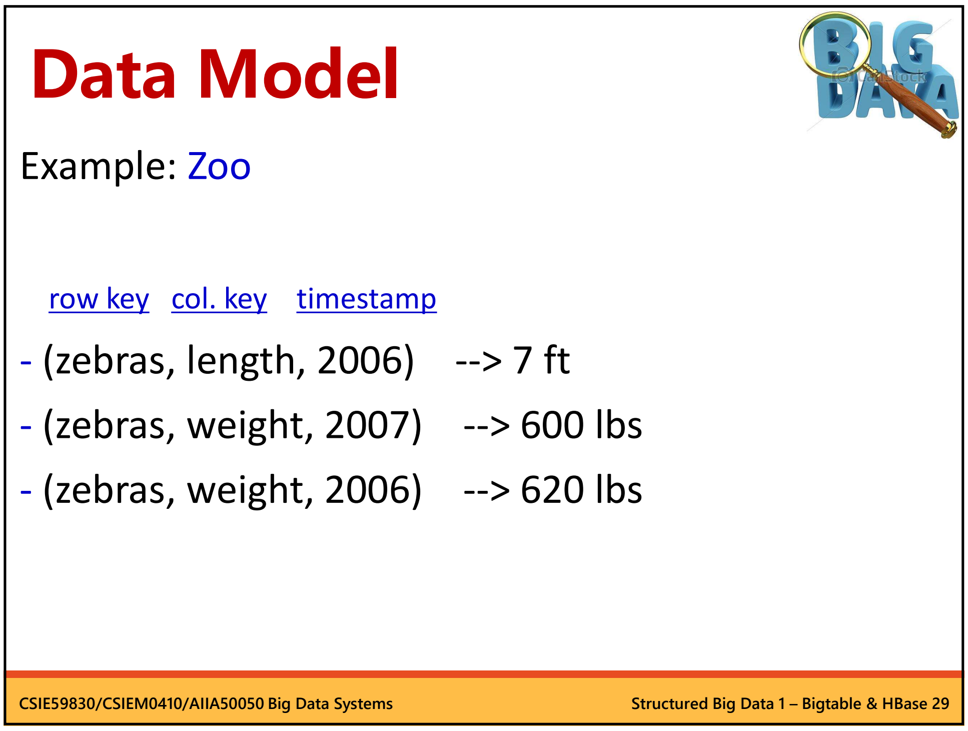
Data Model Example

Example: Zoo

Data Model Example

Example: Zoo

row key col. key timestamp

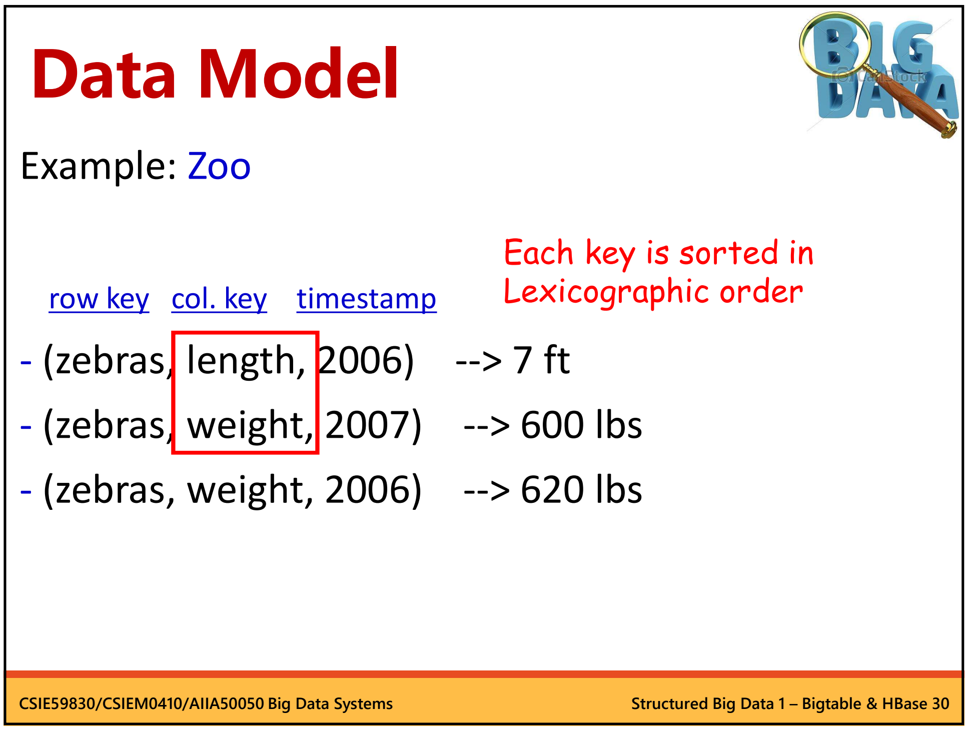


Data Model

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

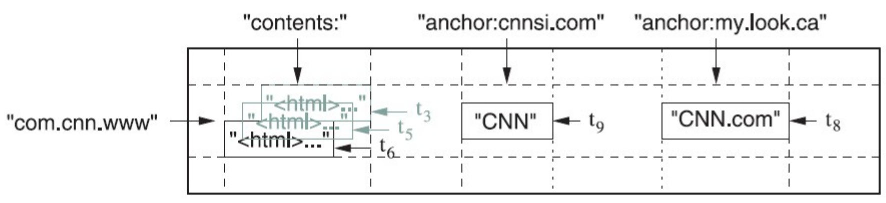
Data Model

Example: Zoo

Timestamp ordering is defined as "most recent appears first"

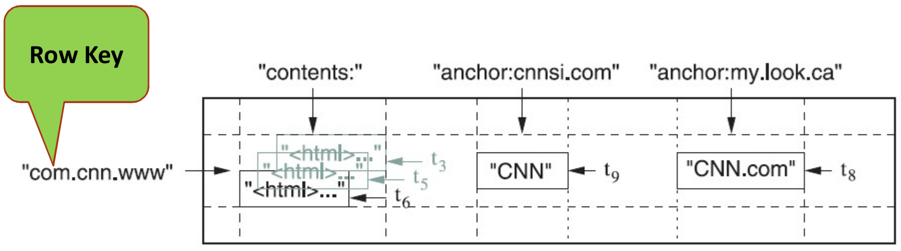
row key	col. key	timestamp	
- (zebras,	length,	2006)	--> 7 ft
- (zebras,	weight,	2007)	--> 600 lbs
- (zebras,	weight,	2006)	--> 620 lbs

Data Model – WebTable Example



A large collection of web pages and related info

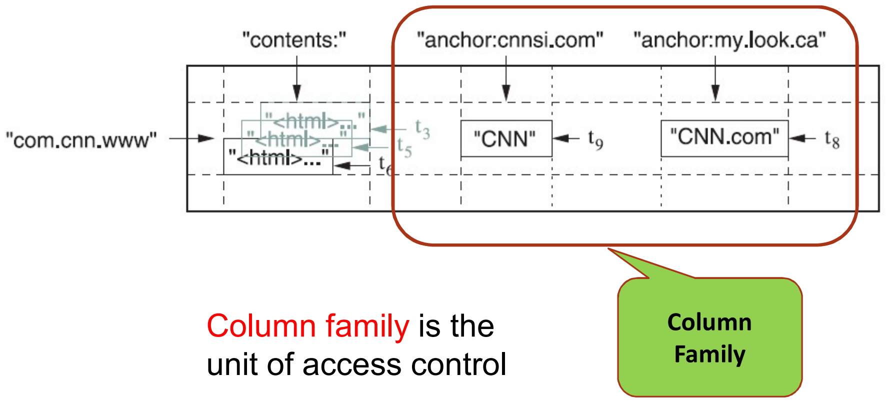
Data Model – WebTable Example



Tablet - Group of rows with consecutive keys.
Unit of Distribution

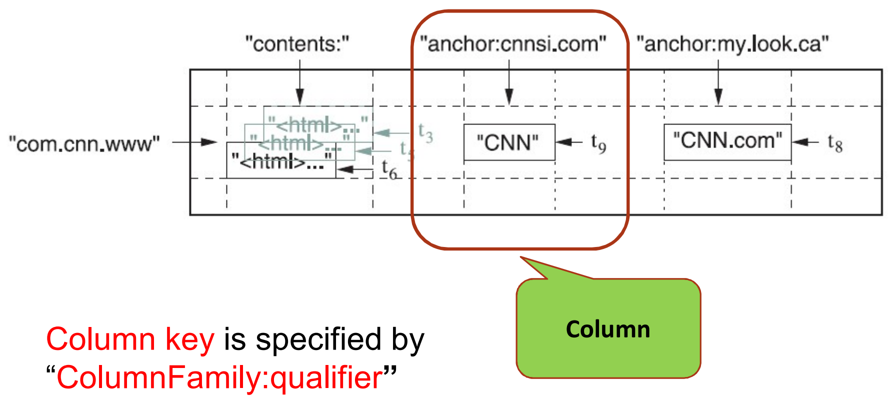
Bigtable maintains data in lexicographic order by row key

Data Model – WebTable Example



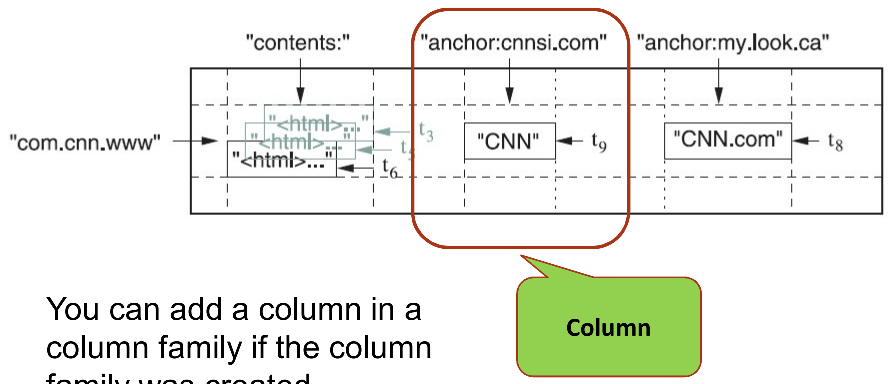
Column family is the unit of access control

Data Model – WebTable Example



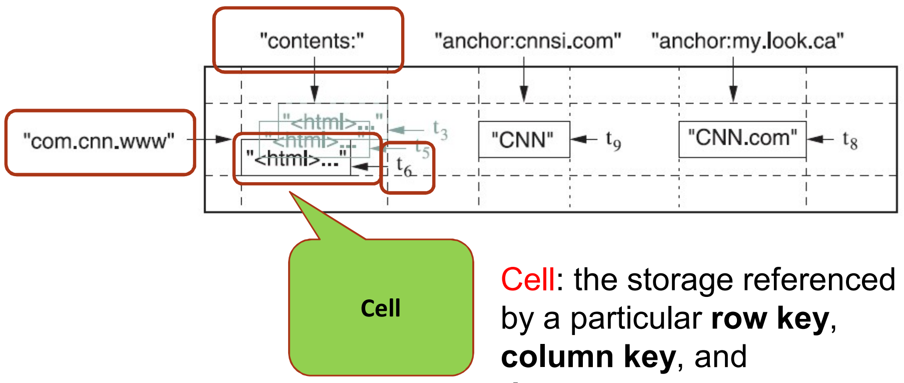
Column key is specified by "ColumnFamily:qualifier"

Data Model – WebTable Example



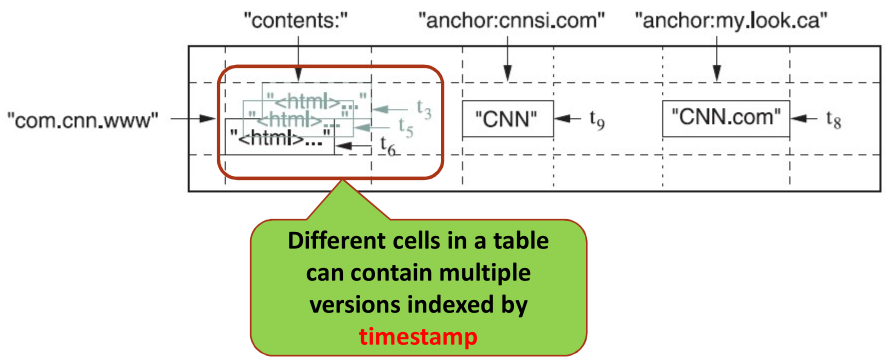
You can add a column in a column family if the column family was created

Data Model – WebTable Example

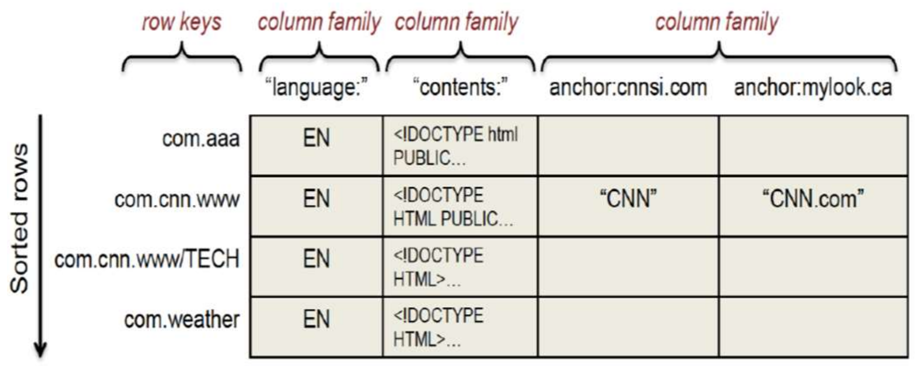


Cell: the storage referenced by a particular **row key**, **column key**, and **timestamp**

Data Model – WebTable Example



A Table Example


Sorted rows

row keys	column family	column family	column family	
	"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>...		

Some Characteristics

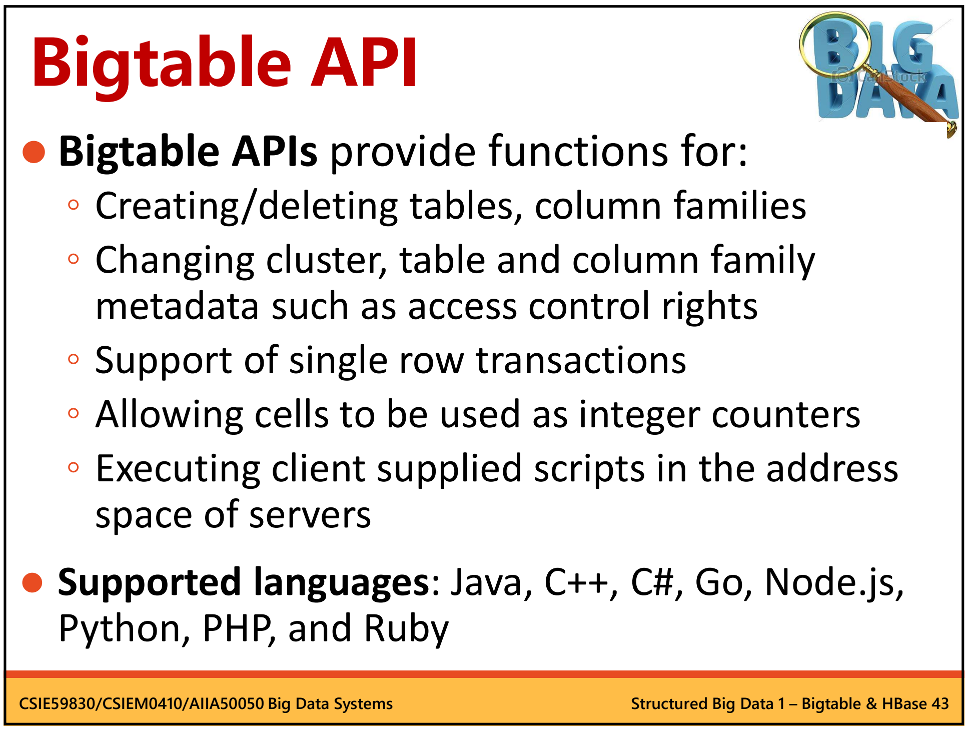
- Each row/column intersection can contain multiple *cells*.
- Each cell contains a unique timestamped version of the data for that row and column.
- A column provides a record of how the stored data has changed over time.
- Columns can be unused in a row.
- Each value is typically no larger than 10 MB.

Bigtable Advantages

- **Large-scale storage** (petabytes of data)
- **High throughput**
- **Low latency** (sub-10ms, millions of requests/second)
- **High availability** (99.999%)
- **High scalability** (table can scale to billions of rows and thousands of columns)
- **Fully-managed** database (simple administration)
- **Cluster resizing without downtime**
- **Integration** with **AI/ML Tools**
- **HBase compatible** (work with Apache ecosystem)

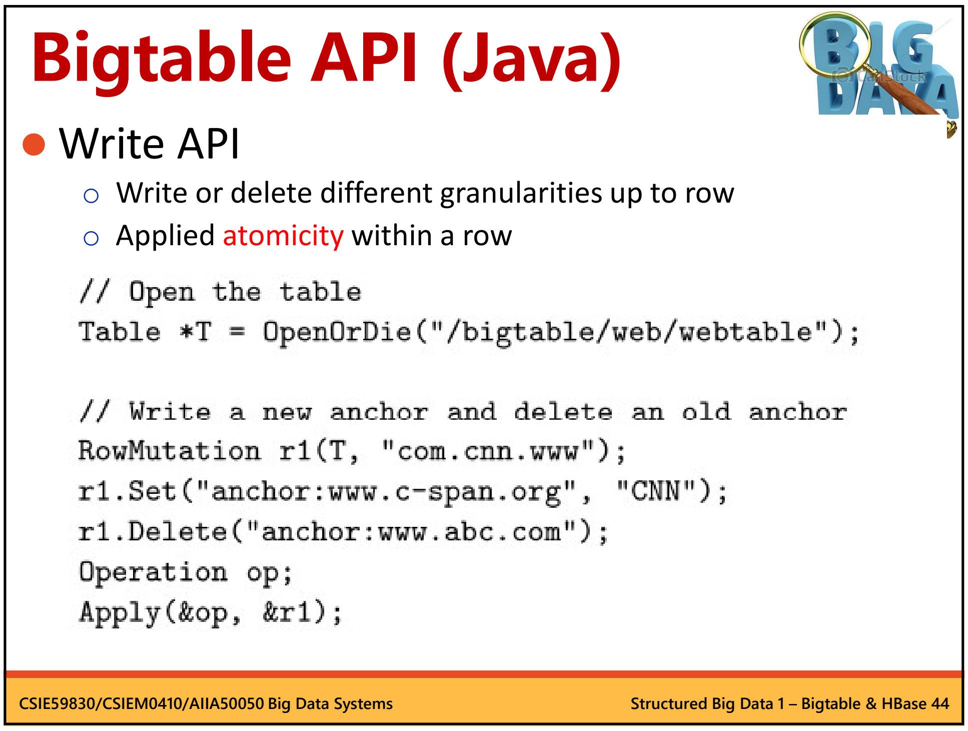
Types of Data/Applications

- Examples of data/applications:
  - Time-series data**,** such as CPU and memory usage over time for multiple servers.
  - Marketing data**,** such as purchase histories and customer preferences.
  - Financial data**,** such as transaction histories, stock prices, and currency exchange rates.
  - Internet of Things data**,** such as usage reports from energy meters and home appliances.
  - Graph data**,** such as information about how users are connected to one another.



Bigtable API

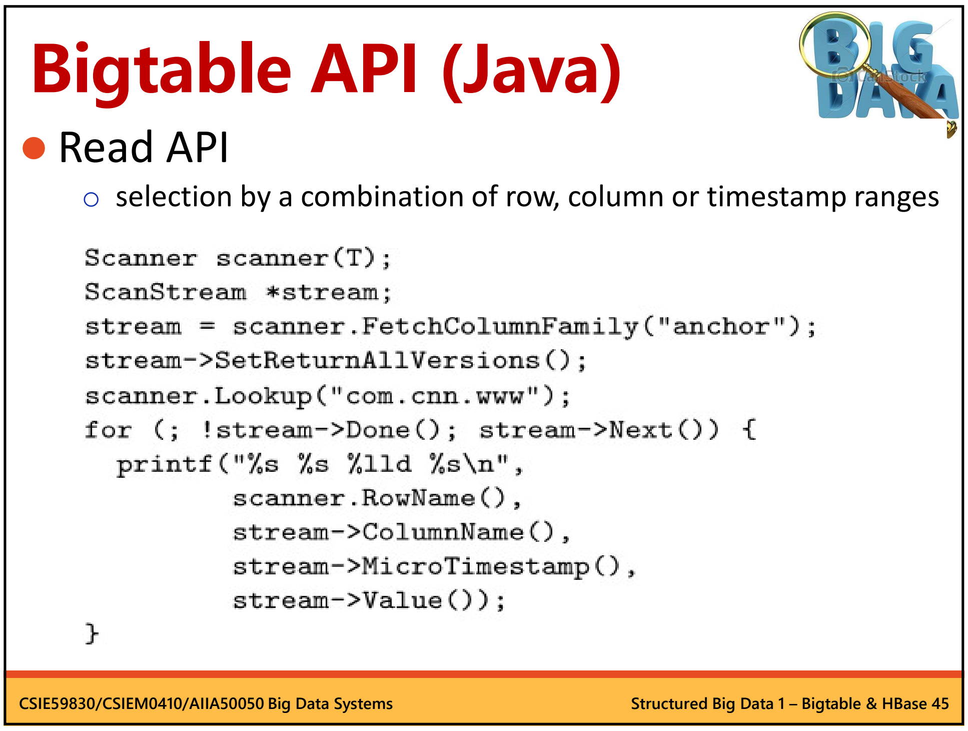
- **Bigtable APIs** provide functions for:
  - Creating/deleting tables, column families
  - Changing cluster, table and column family metadata such as access control rights
  - Support of single row transactions
  - Allowing cells to be used as integer counters
  - Executing client supplied scripts in the address space of servers
- **Supported languages**: Java, C++, C#, Go, Node.js, Python, PHP, and Ruby



Bigtable API (Java)

- Write API
  - Write or delete different granularities up to row
  - Applied atomicity within a row

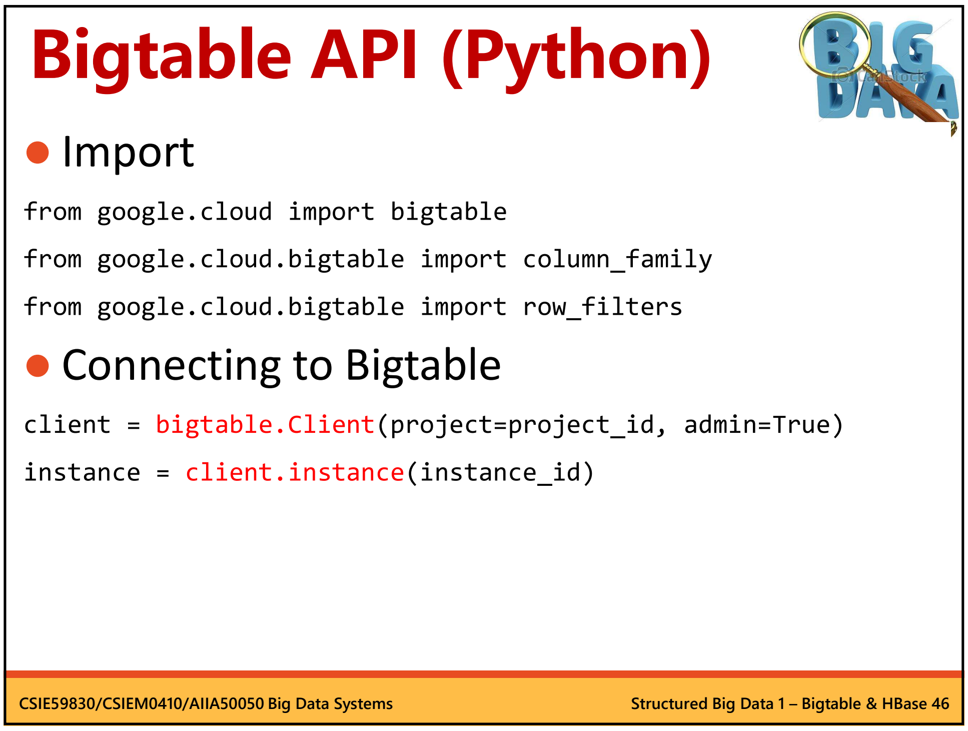
```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// 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 (Java)

- 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());
}
```



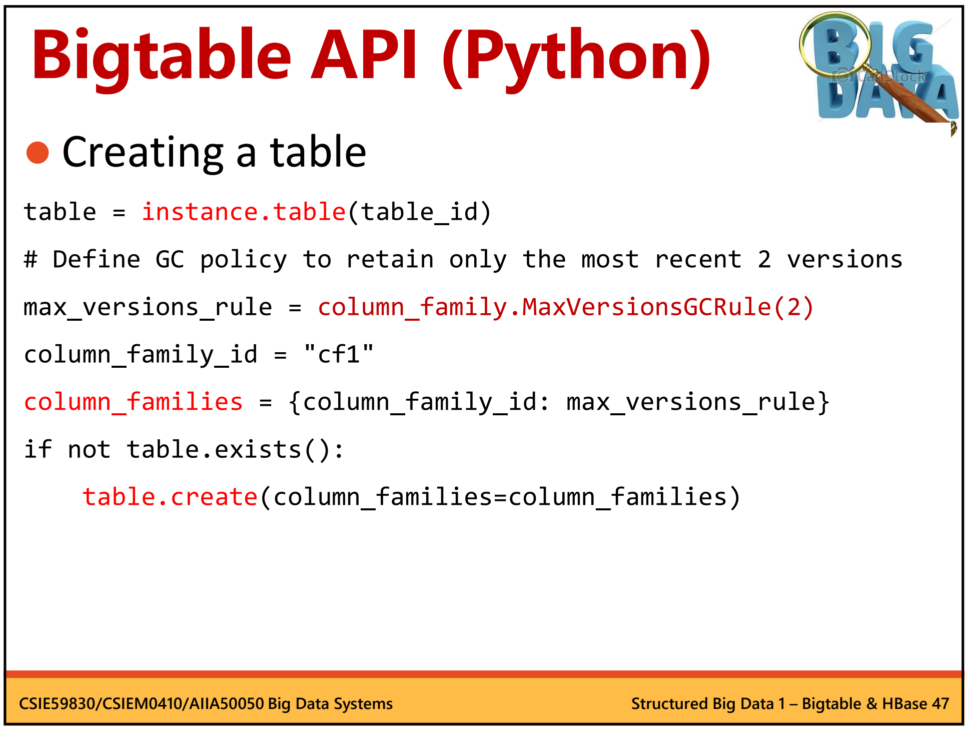
Bigtable API (Python)

- Import

```
from google.cloud import bigtable
from google.cloud.bigtable import column_family
from google.cloud.bigtable import row_filters
```

- Connecting to Bigtable

```
client = bigtable.Client(project=project_id, admin=True)
instance = client.instance(instance_id)
```

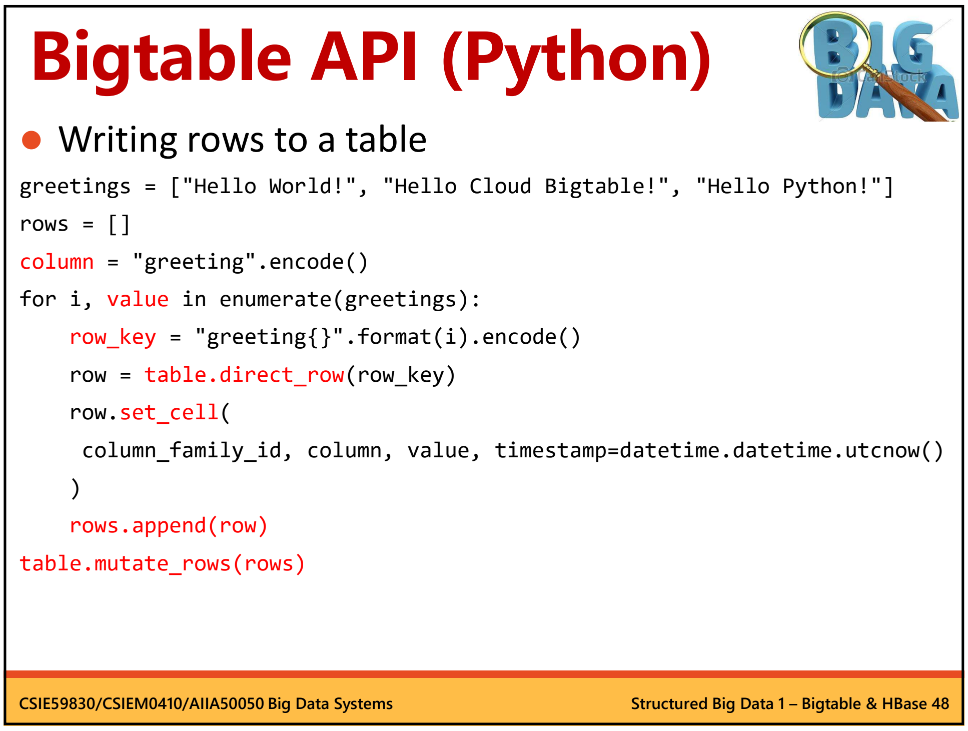


Bigtable API (Python)

- Creating a table

```
table = instance.table(table_id)
# Define GC policy to retain only the most recent 2 versions
max_versions_rule = column_family.MaxVersionsGCRule(2)
column_family_id = "cf1"
column_families = {column_family_id: max_versions_rule}
if not table.exists():
    table.create(column_families=column_families)
```

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 47



Bigtable API (Python)

- Writing rows to a table

```
greetings = ["Hello World!", "Hello Cloud Bigtable!", "Hello Python!"]
rows = []
column = "greeting".encode()
for i, value in enumerate(greetings):
    row_key = "greeting{}".format(i).encode()
    row = table.direct_row(row_key)
    row.set_cell(
     column_family_id, column, value, timestamp=datetime.datetime.utcnow()
    )
    rows.append(row)
table.mutate_rows(rows)
```

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 48

Bigtable API (Python)

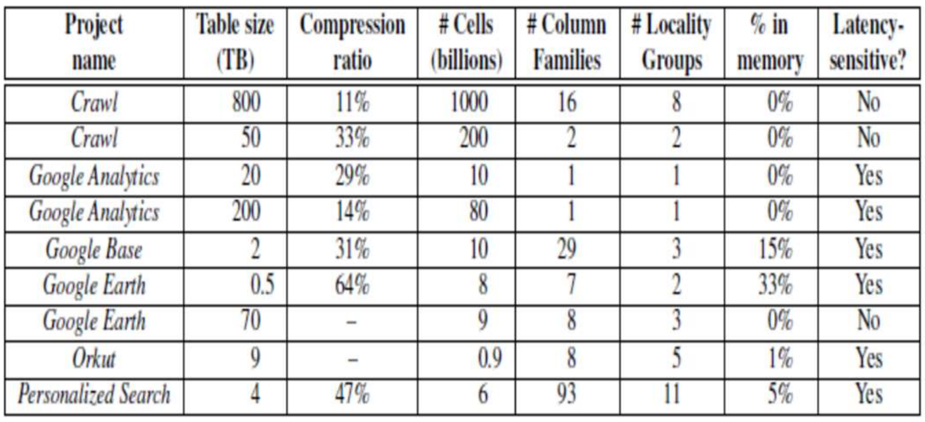
- Reading a row by key and filter

```
row_filter = row_filters.CellsColumnLimitFilter(1)
key = "greeting0".encode()
row = table.read_row(key, row_filter)
cell = row.cells[column_family_id][column][0]
print(cell.value.decode("utf-8"))
```

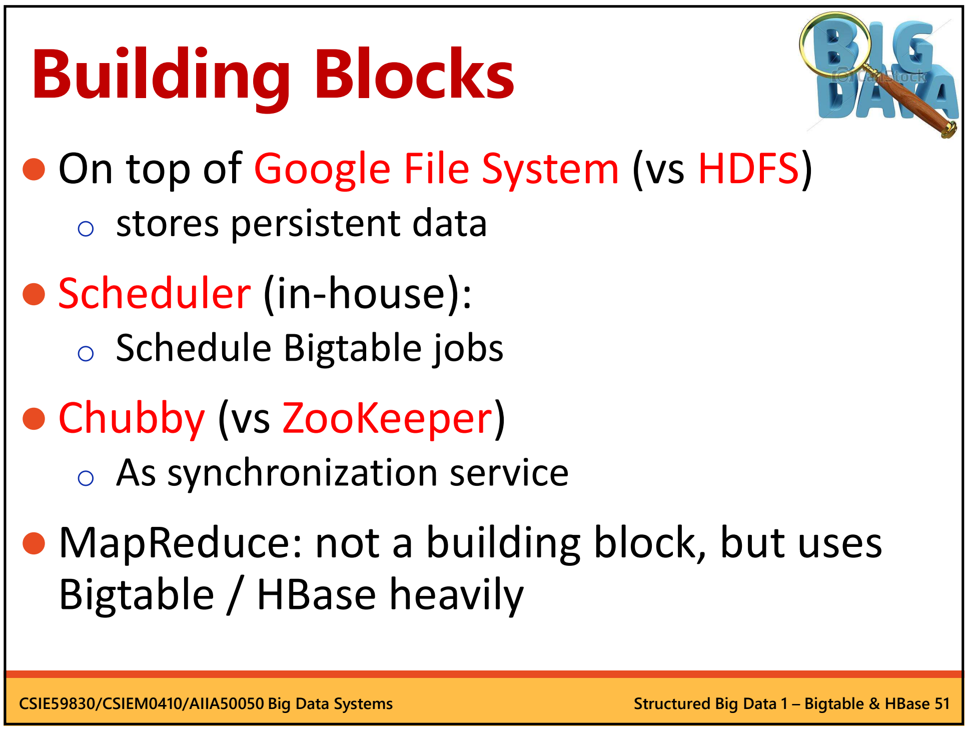
- Scanning all rows

```
partial_rows = table.read_rows(filter_=row_filter)
for row in partial_rows:
    cell = row.cells[column_family_id][column][0]
    print(cell.value.decode("utf-8"))
```

Google Applications using BigTable

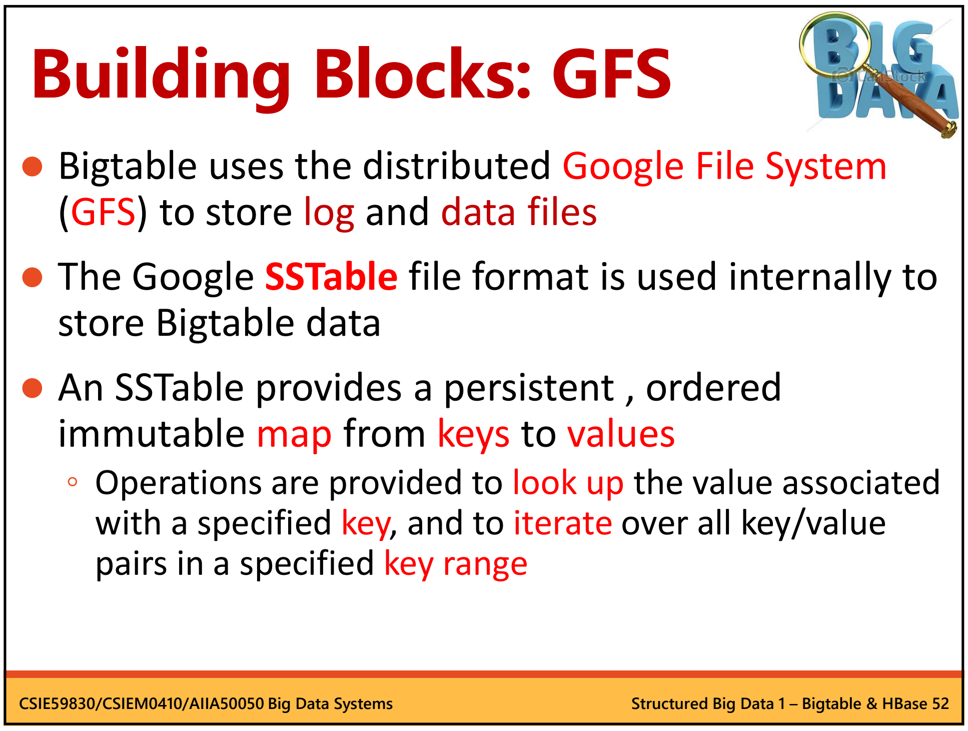


Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
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



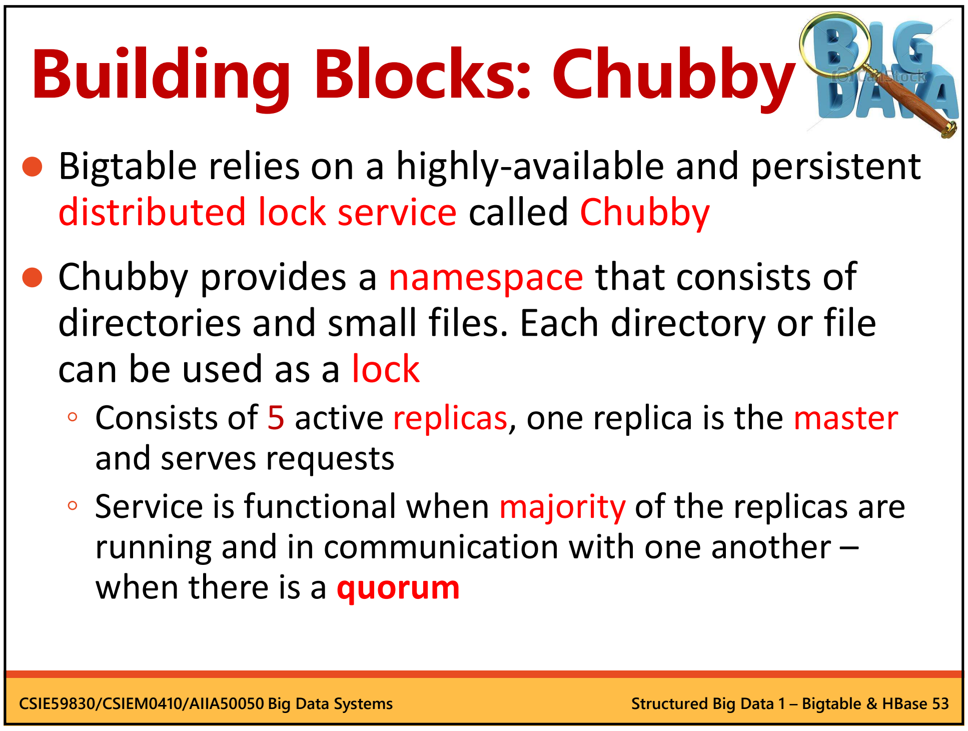
Building Blocks

- 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



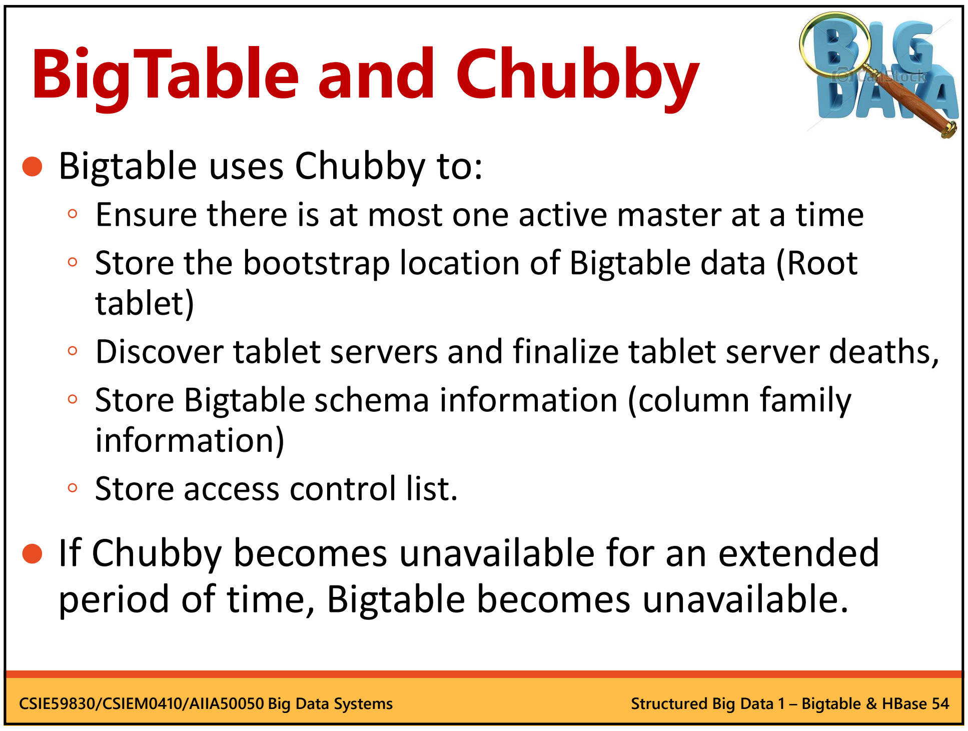
Building Blocks: GFS

- 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



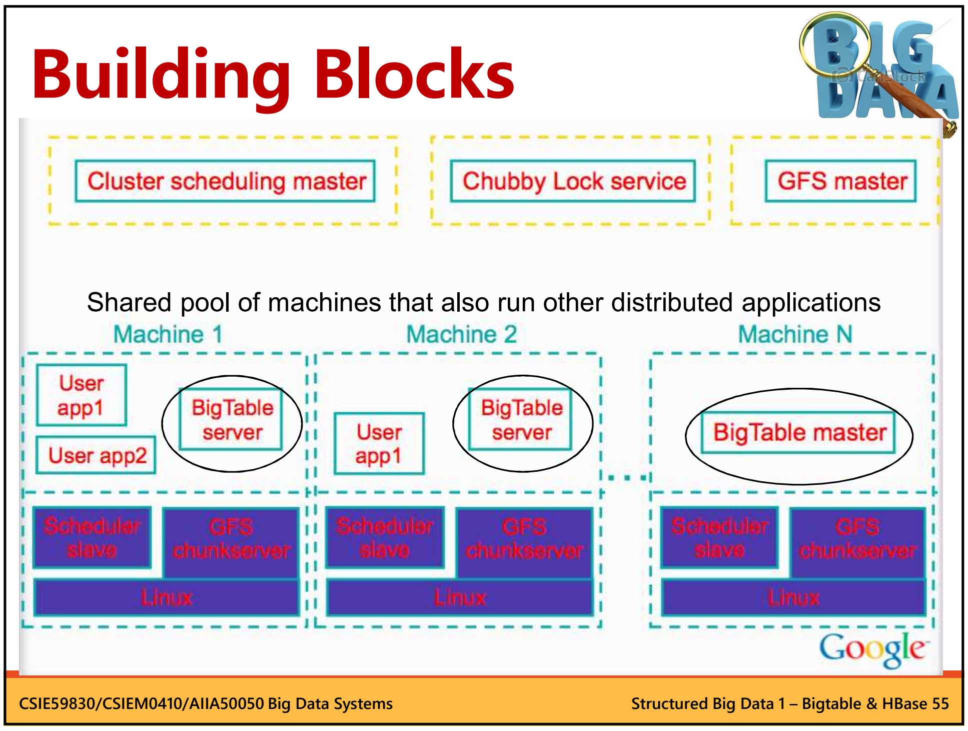
Building Blocks: Chubby

- 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**

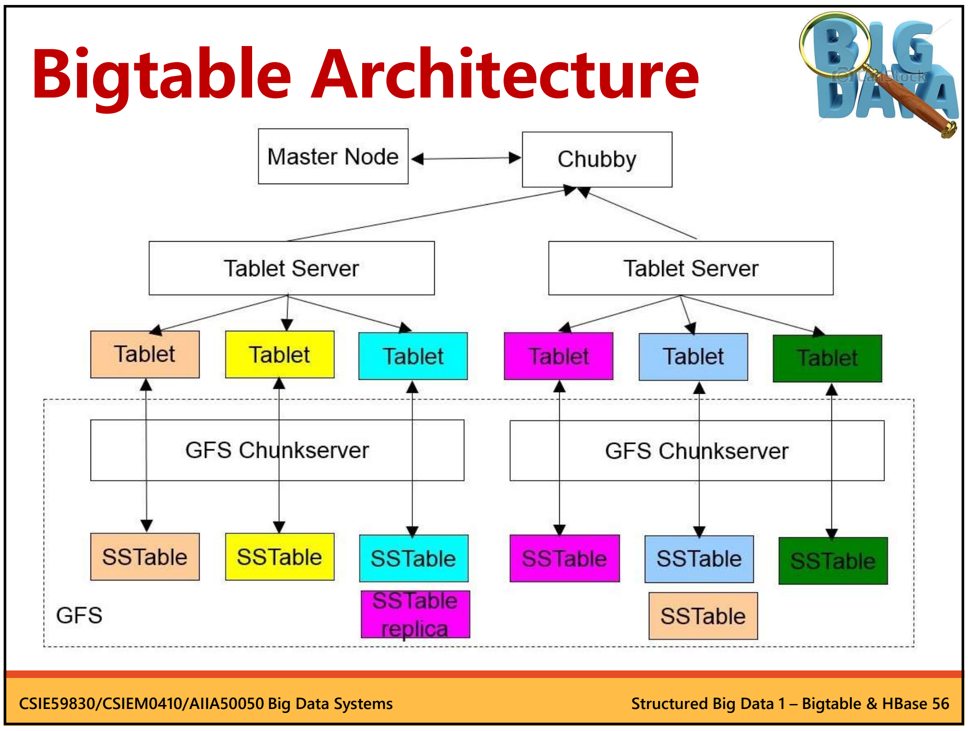


BigTable and Chubby

- 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.



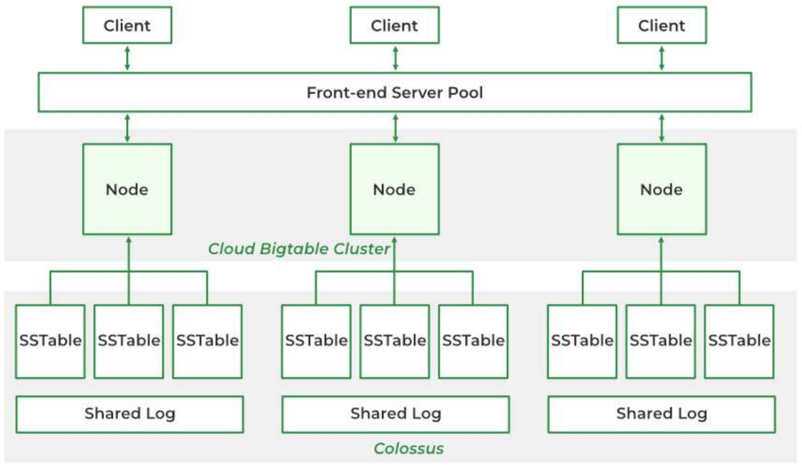
Building Blocks



Bigtable Architecture

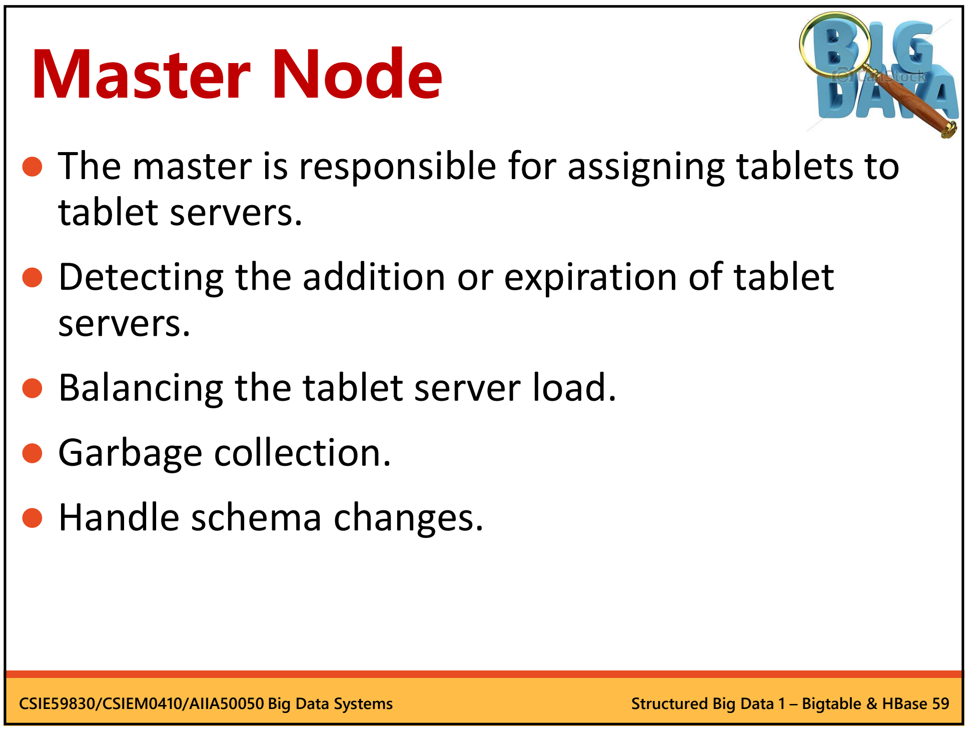
Arch on top of Colossus

- Current Cloud Bigtable is built on top of Colossus.



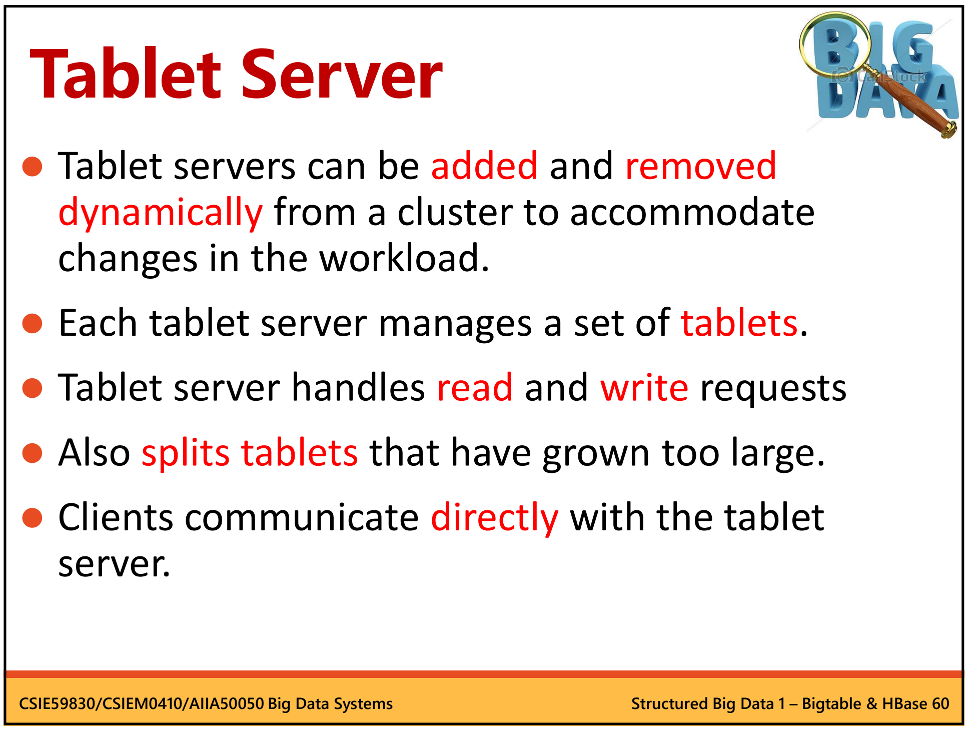
Organization

- 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



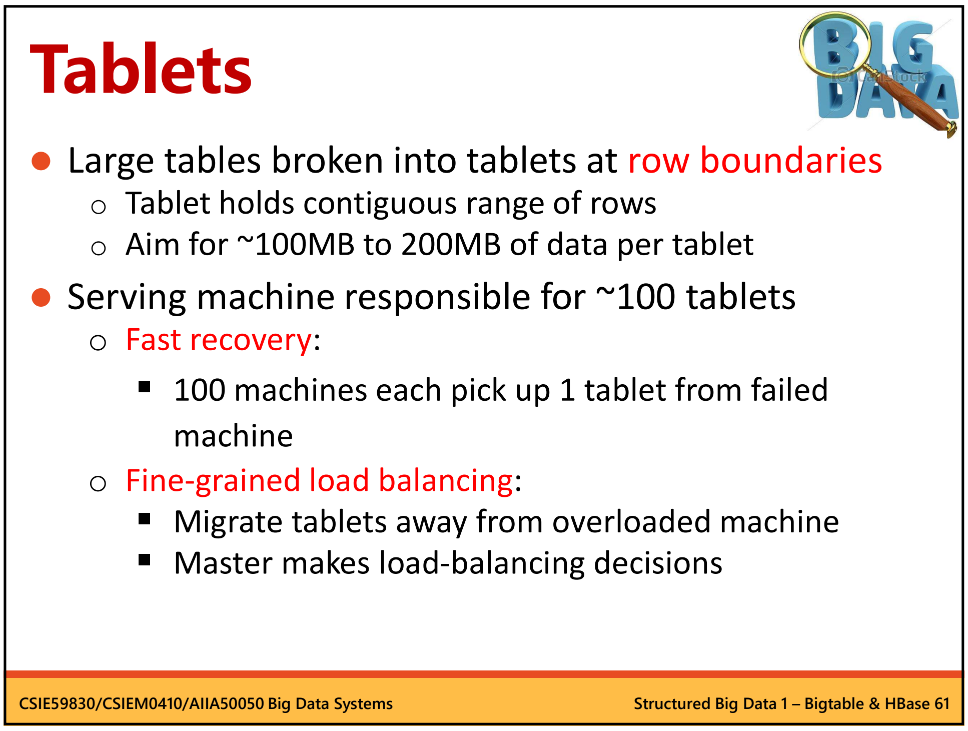
Master Node

- The master is responsible for assigning tablets to tablet servers.
- Detecting the addition or expiration of tablet servers.
- Balancing the tablet server load.
- Garbage collection.
- Handle schema changes.



Tablet Server

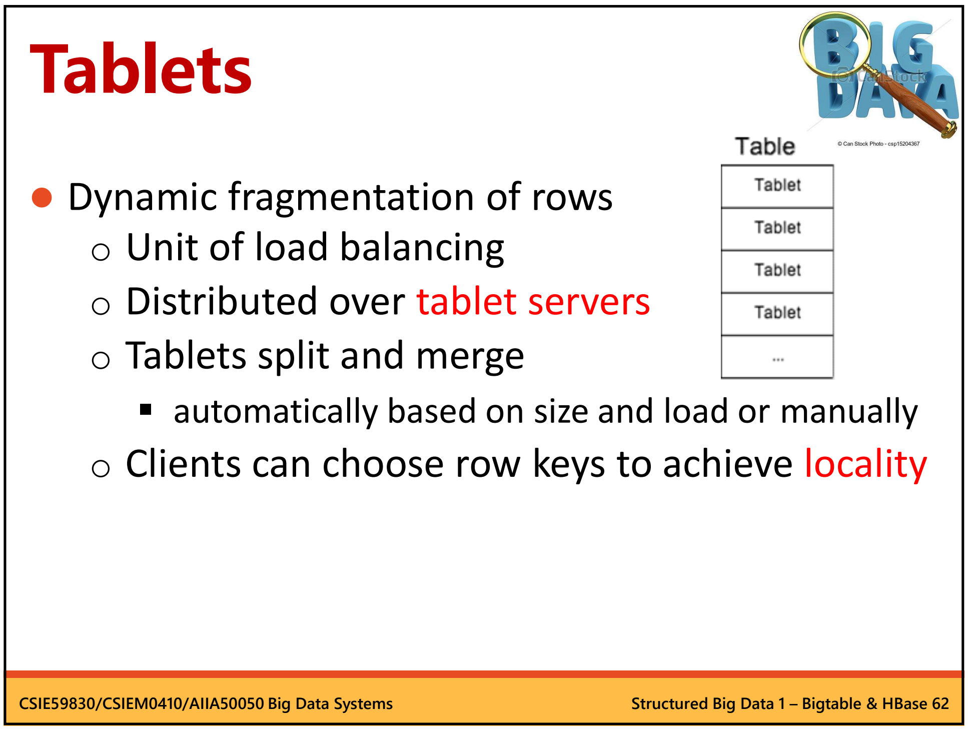
- Tablet servers can be added and removed dynamically from a cluster to accommodate changes in the workload.
- Each tablet server manages a set of tablets.
- Tablet server handles read and write requests
- Also splits tablets that have grown too large.
- Clients communicate directly with the tablet server.



Tablets

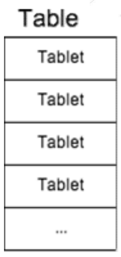
- Large tables broken into tablets at row boundaries
  - Tablet holds contiguous range of rows
  - Aim for ~100MB to 200MB of data per tablet
- Serving machine responsible for ~100 tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing: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

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 61

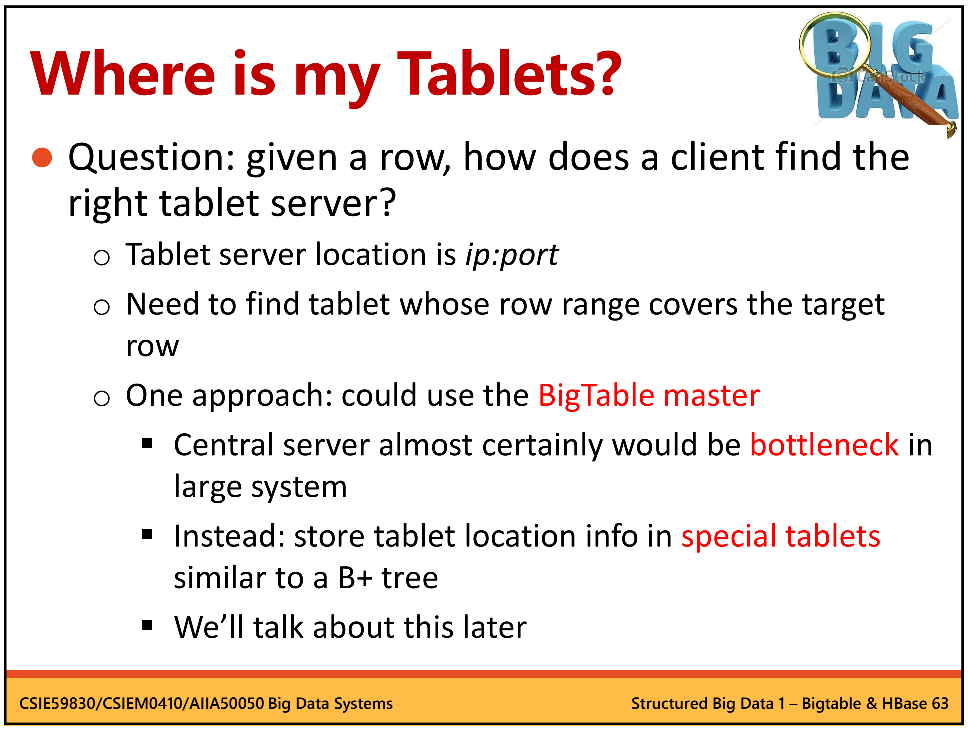


Tablets

- 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

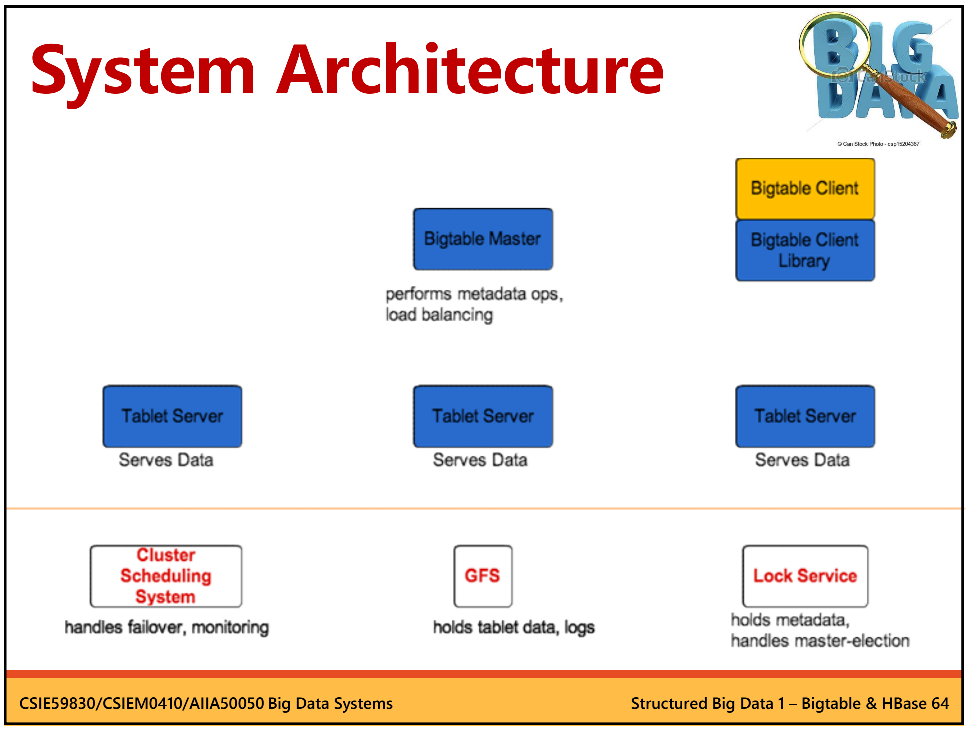


CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 62



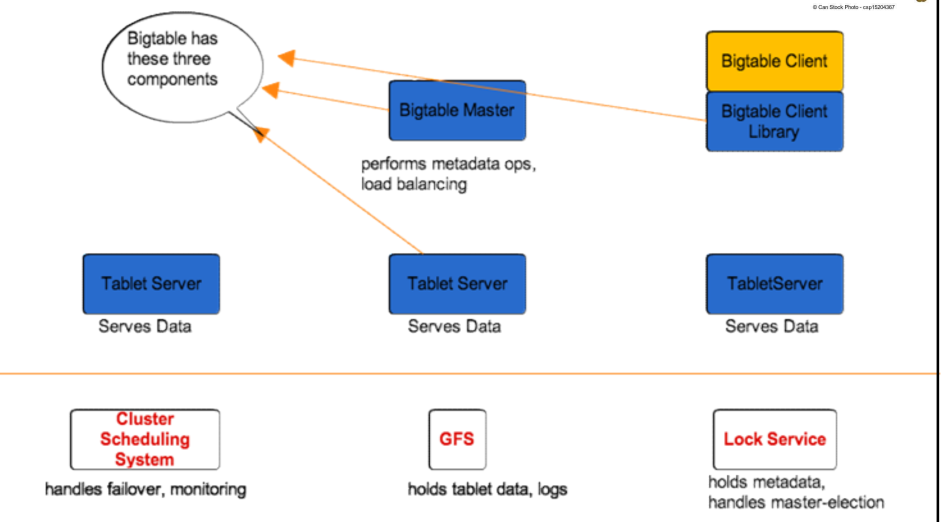
Where is my Tablets?

- 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

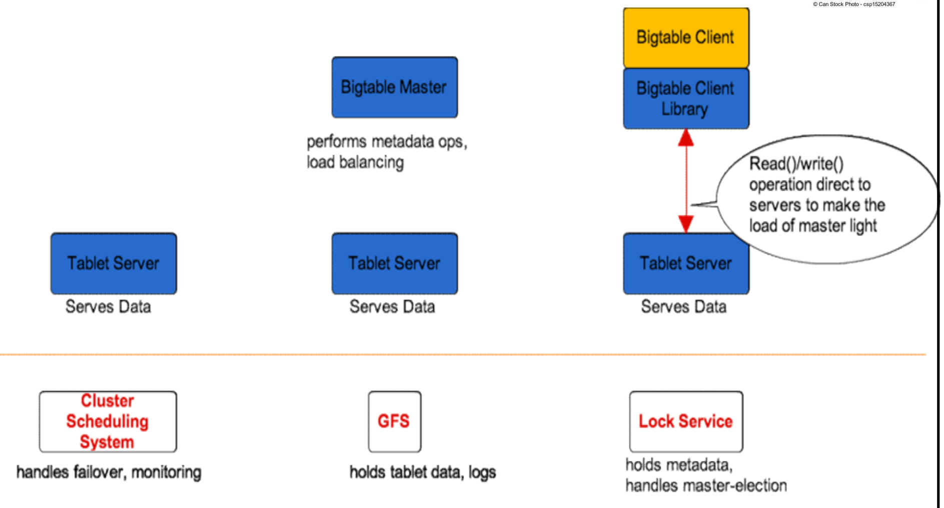


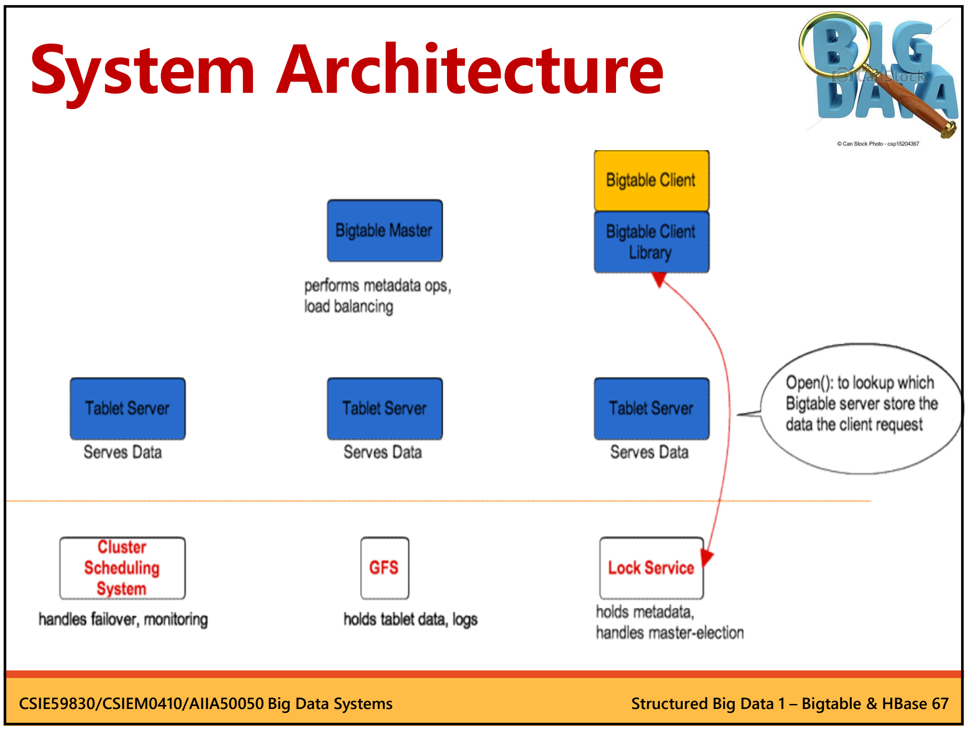
System Architecture

System Architecture

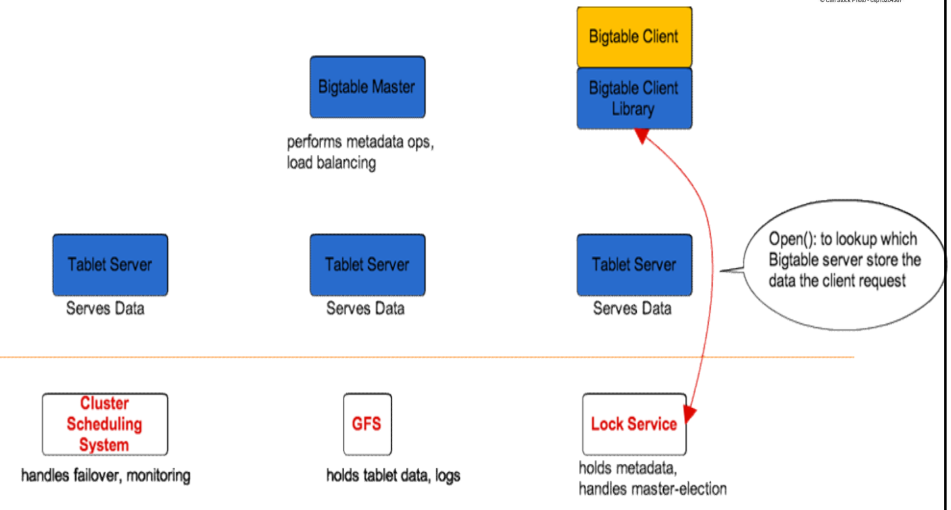

Bigtable has these three components
Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Tablet Server
Serves Data
Tablet Server
Serves Data
TabletServer
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election

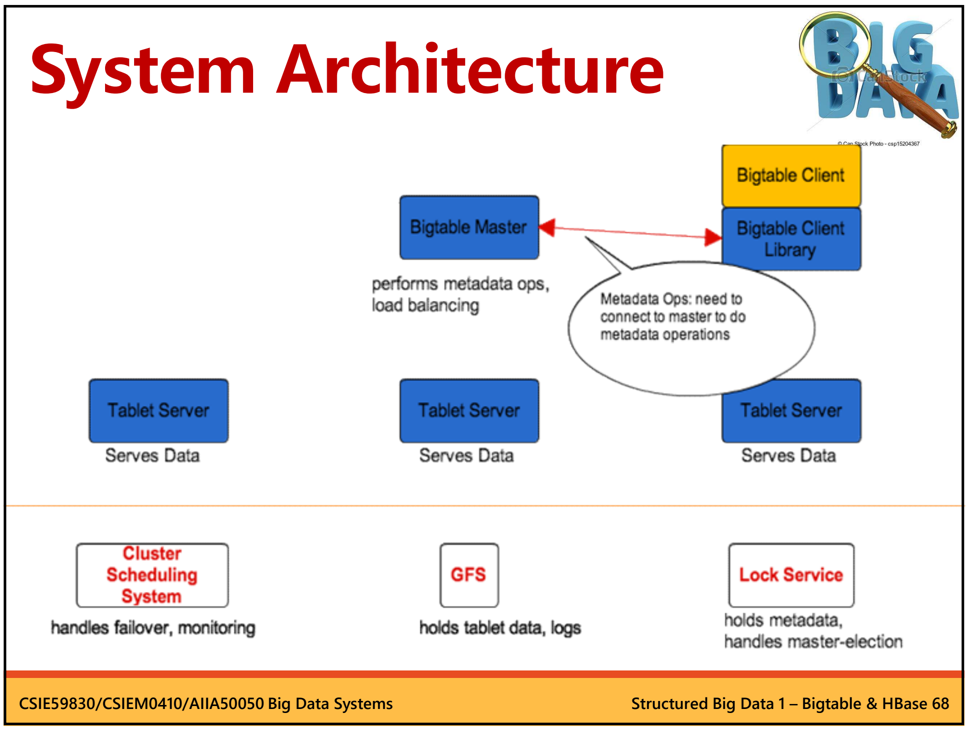
System Architecture


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

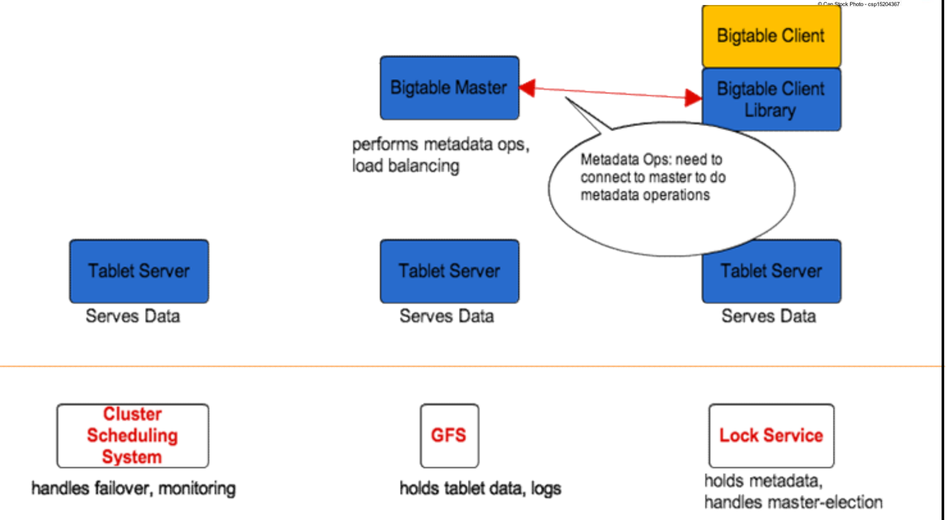


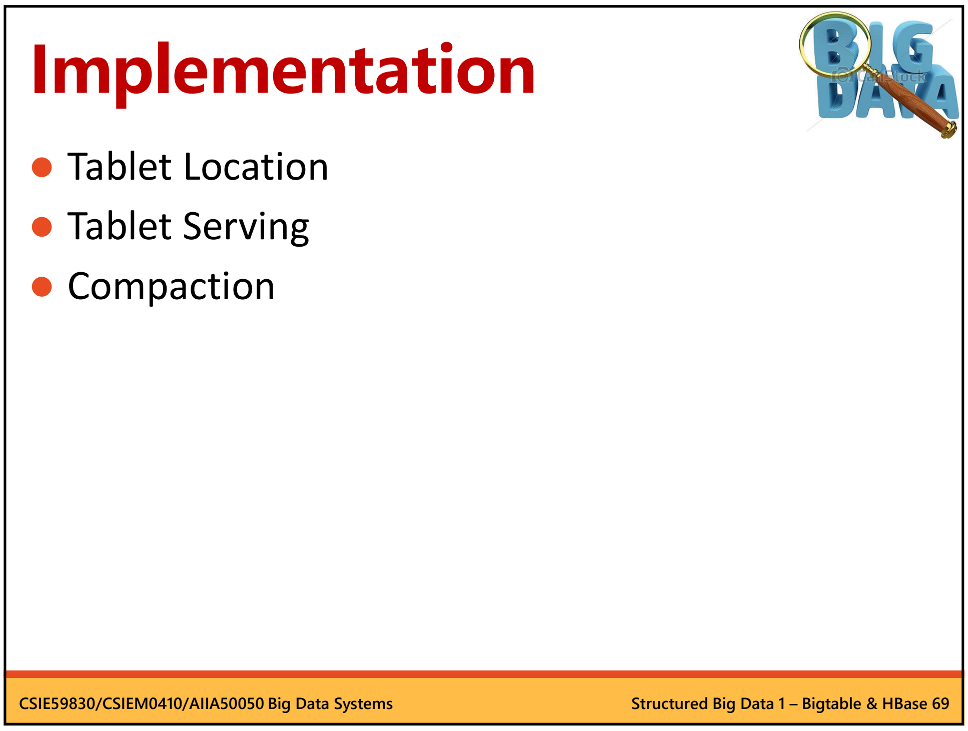
System Architecture


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



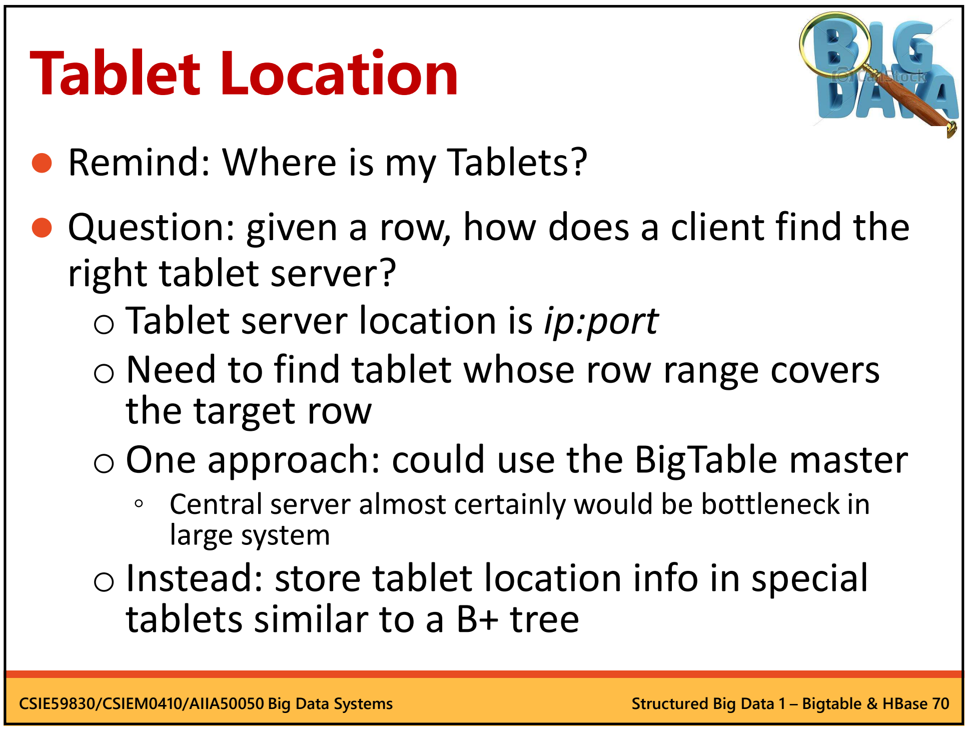
System Architecture


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



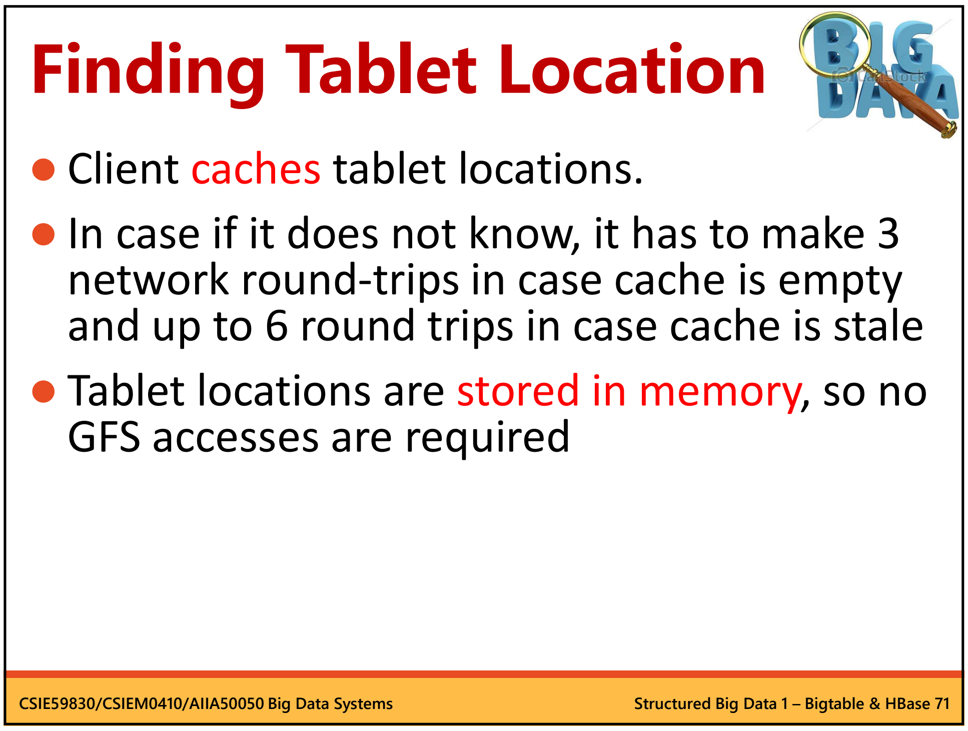
Implementation

- Tablet Location
- Tablet Serving
- Compaction



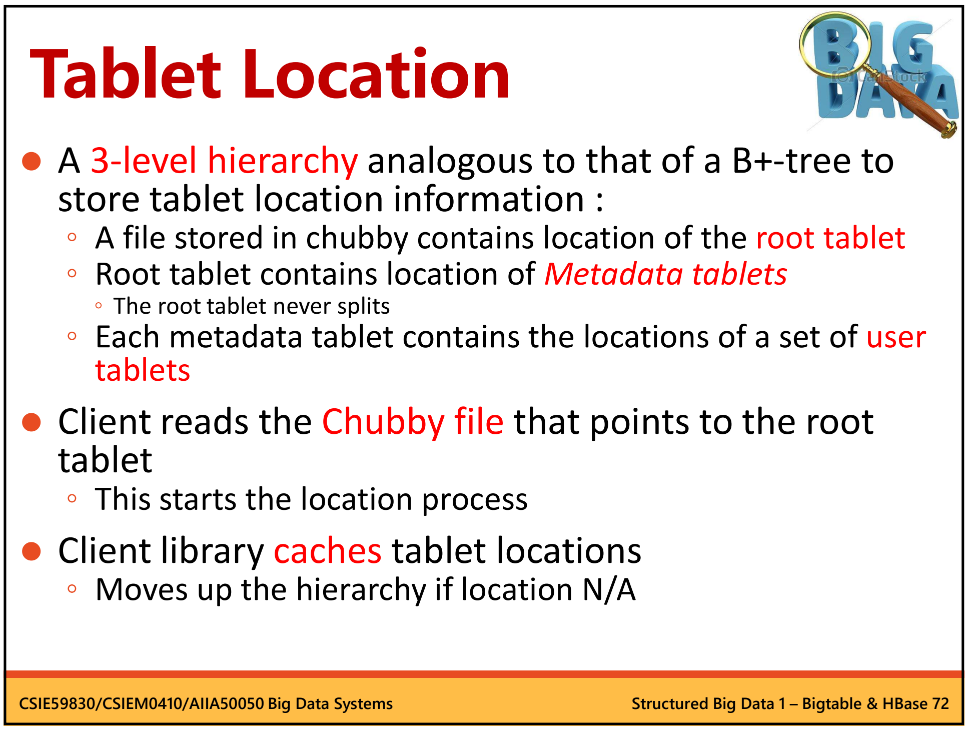
Tablet Location

- 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



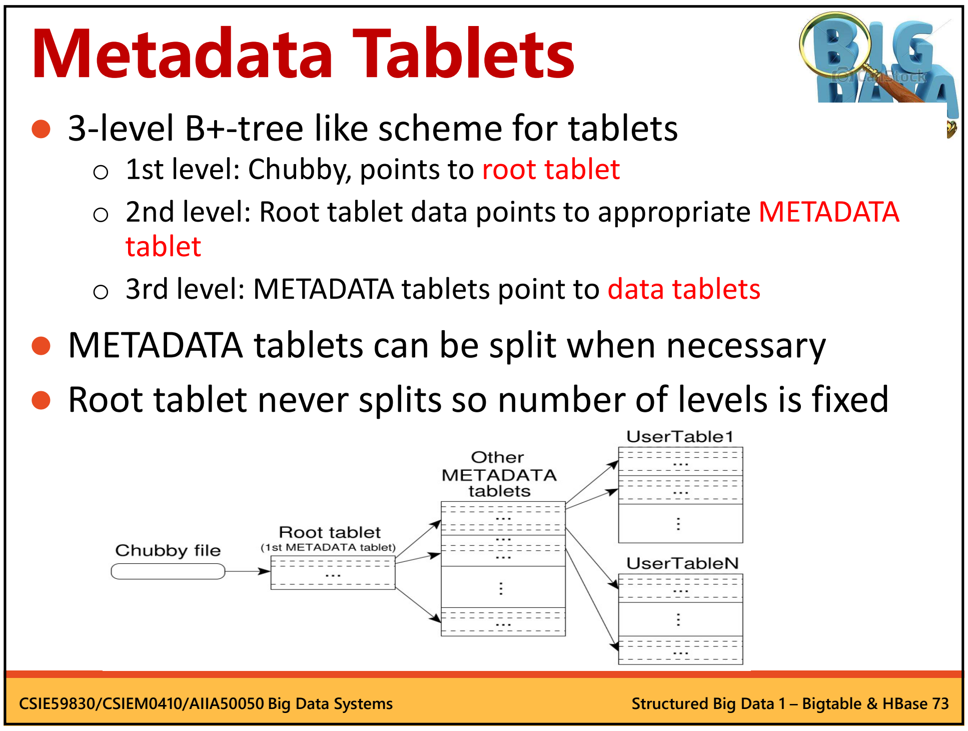
Finding Tablet Location

- Client caches tablet locations.
- In case if it does not know, it has to make 3 network round-trips in case cache is empty and up to 6 round trips in case cache is stale
- Tablet locations are stored in memory, so no GFS accesses are required



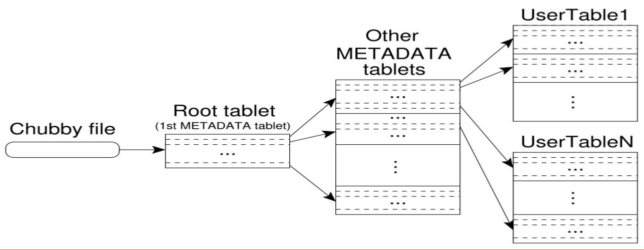
Tablet Location

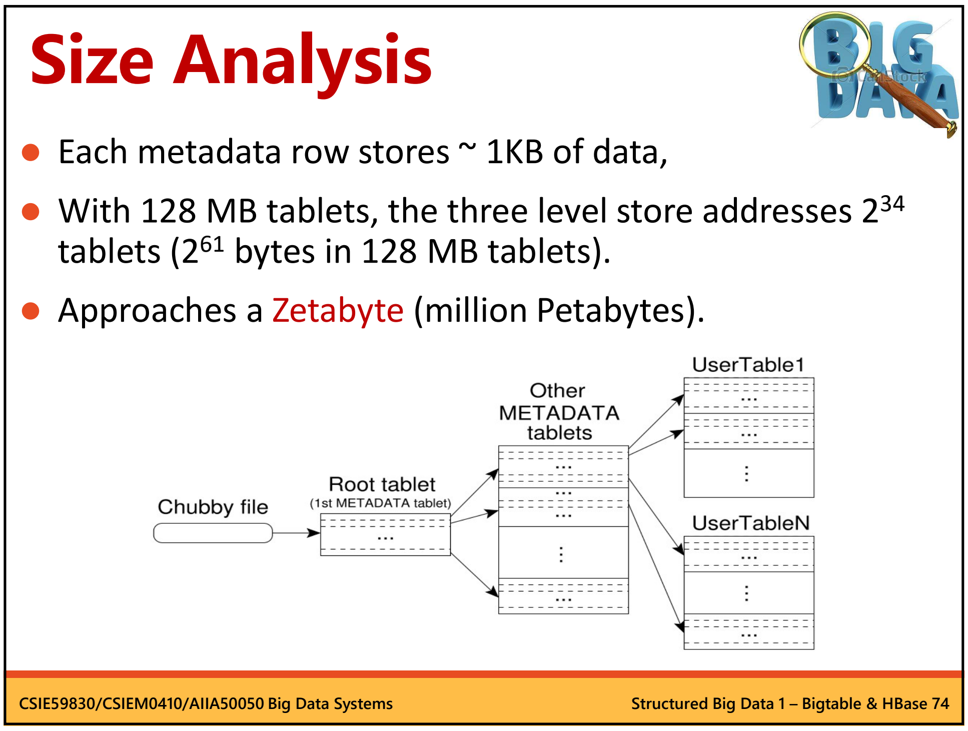
- A 3-level hierarchy analogous to that of a B+-tree to store tablet location information :
  - A file stored in chubby contains location of the root tablet
  - Root tablet contains location of *Metadata tablets*
    - The root tablet never splits
  - Each metadata tablet contains the locations of a 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

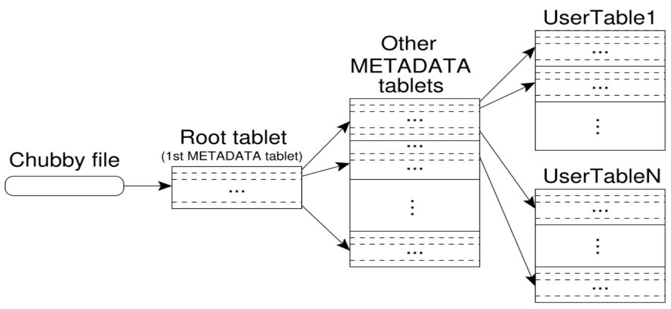
- 3-level B+-tree like scheme for tablets
  - 1st level: Chubby, points to root tablet
  - 2nd level: Root tablet data points to appropriate METADATA tablet
  - 3rd level: METADATA tablets point to data tablets
- METADATA tablets can be split when necessary
- Root tablet never splits so number of levels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

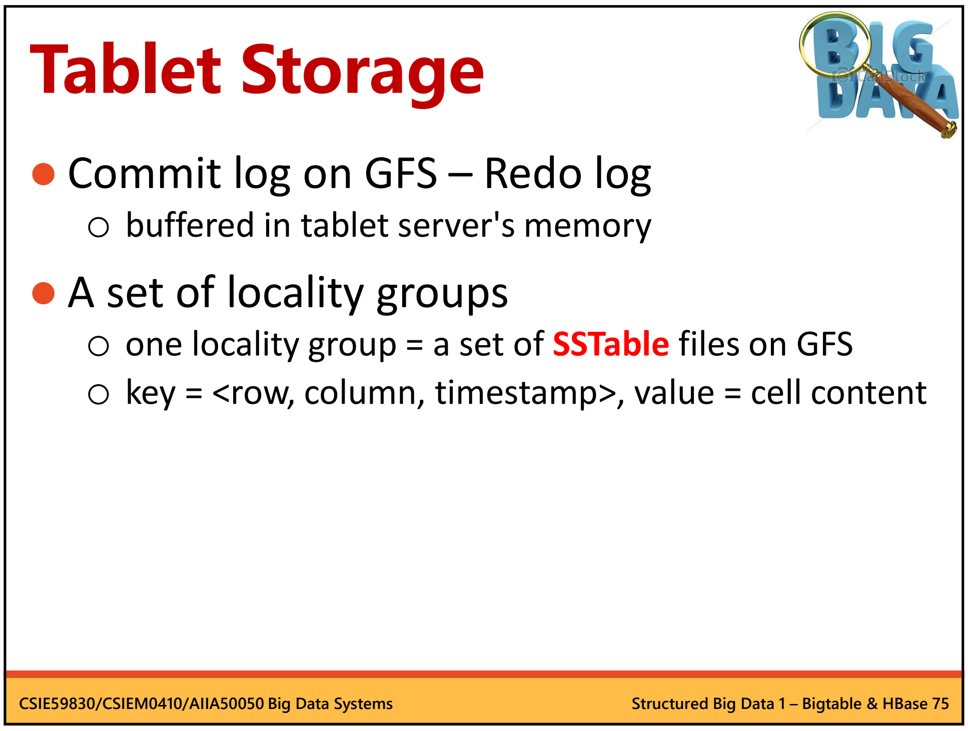




Size Analysis

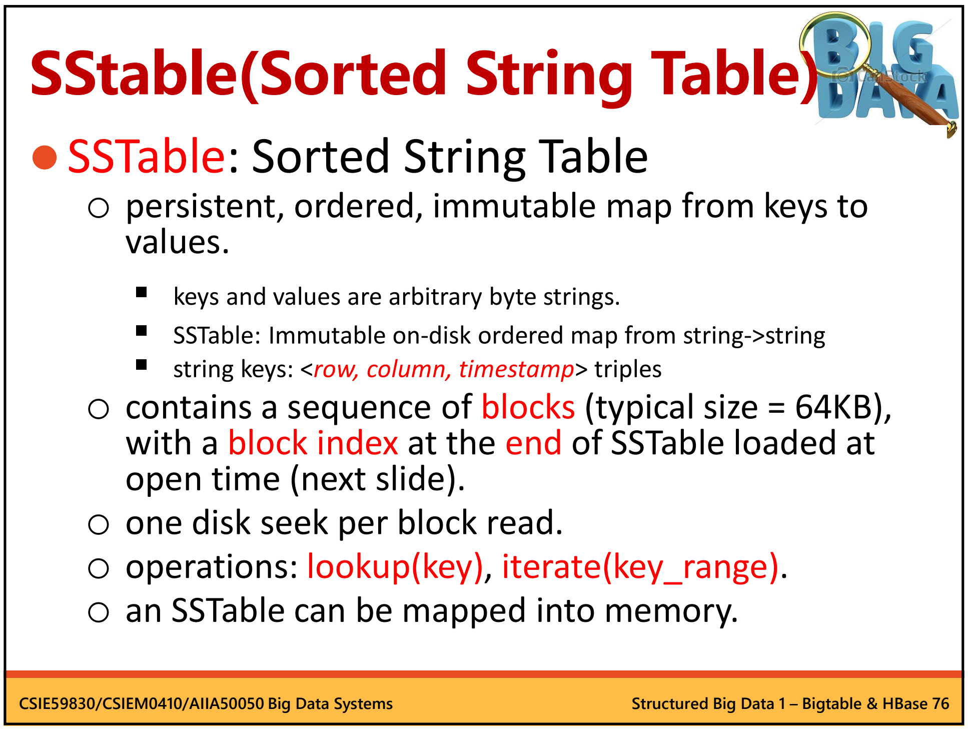
- Each metadata row stores ~ 1KB of data,
- With 128 MB tablets, the three level store addresses 2^{34} tablets (2^{61} bytes in 128 MB tablets).
- Approaches a Zetabyte (million Petabytes).





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

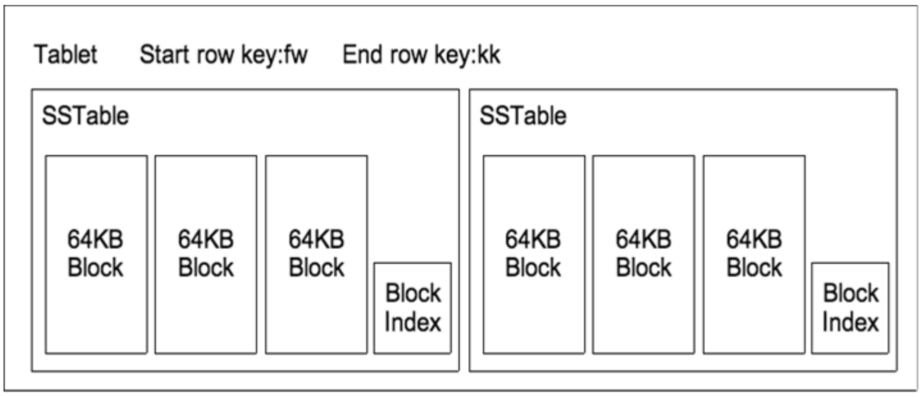


SStable(Sorted String Table)

- 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.

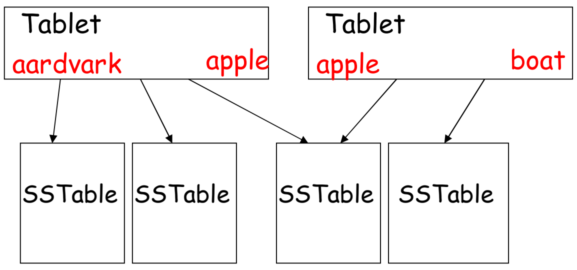
Tablet

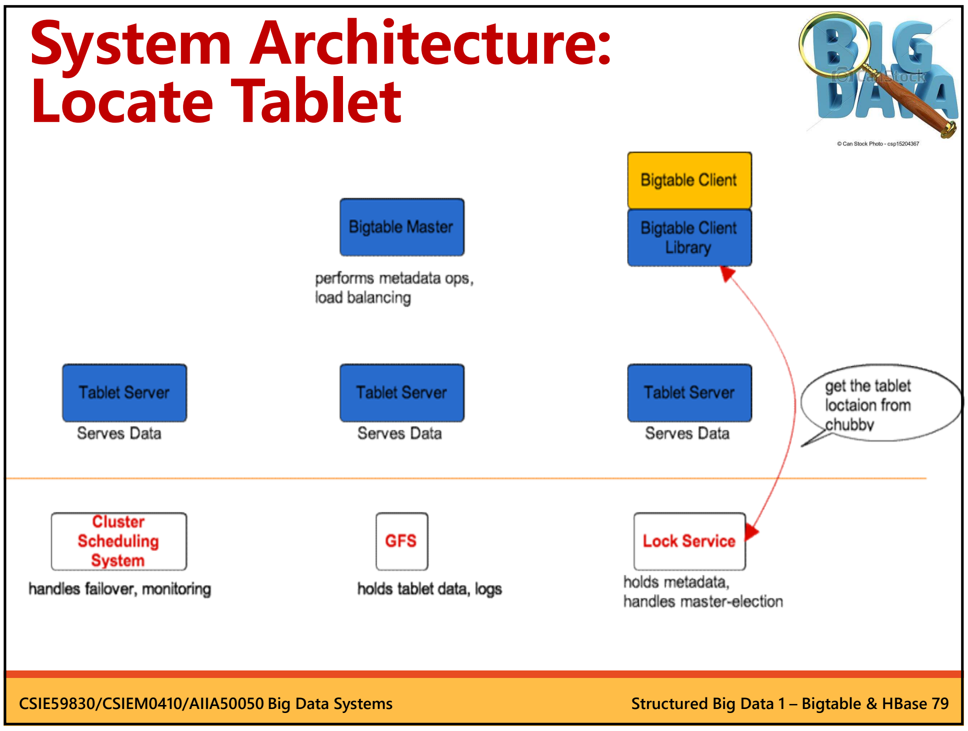
- Contains some range of rows of the table
- Built out of multiple SSTables



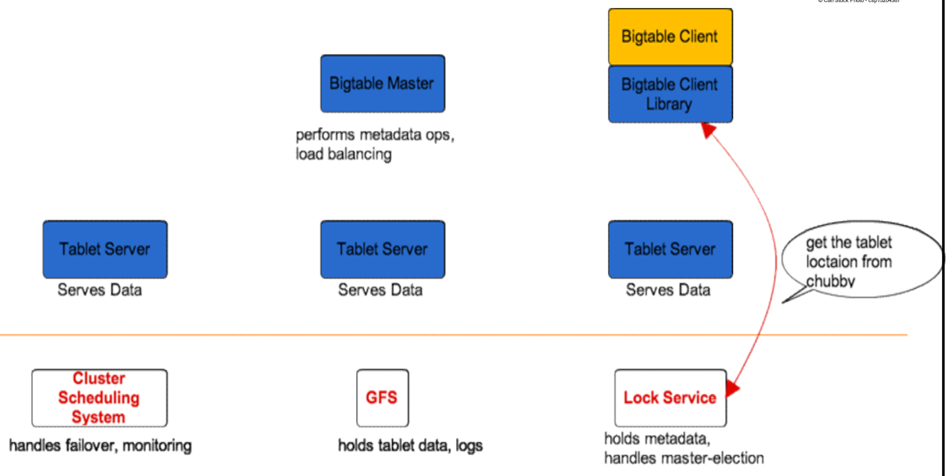
Table

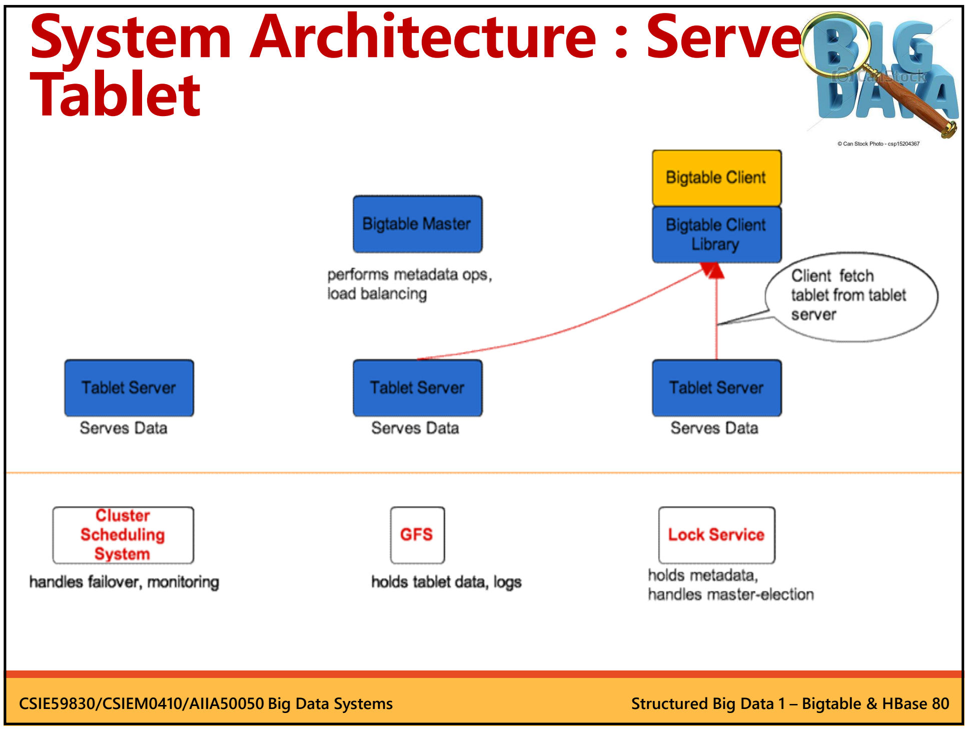
- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



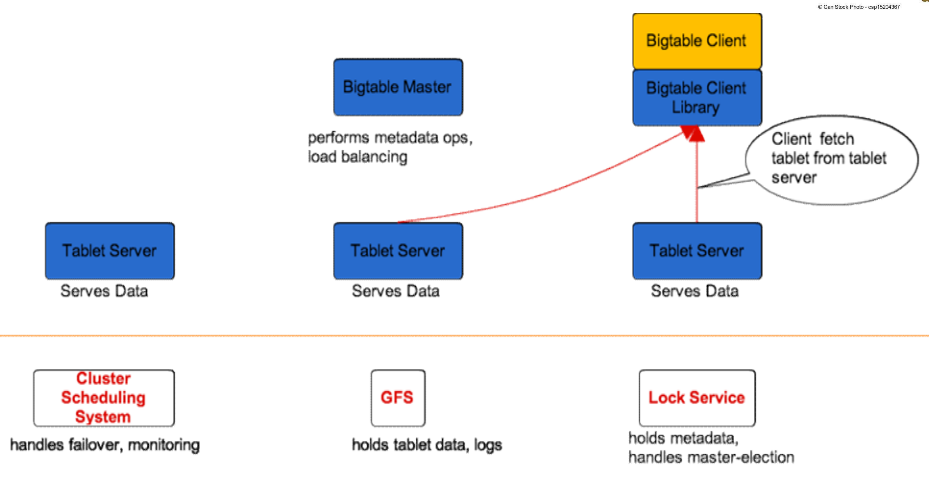


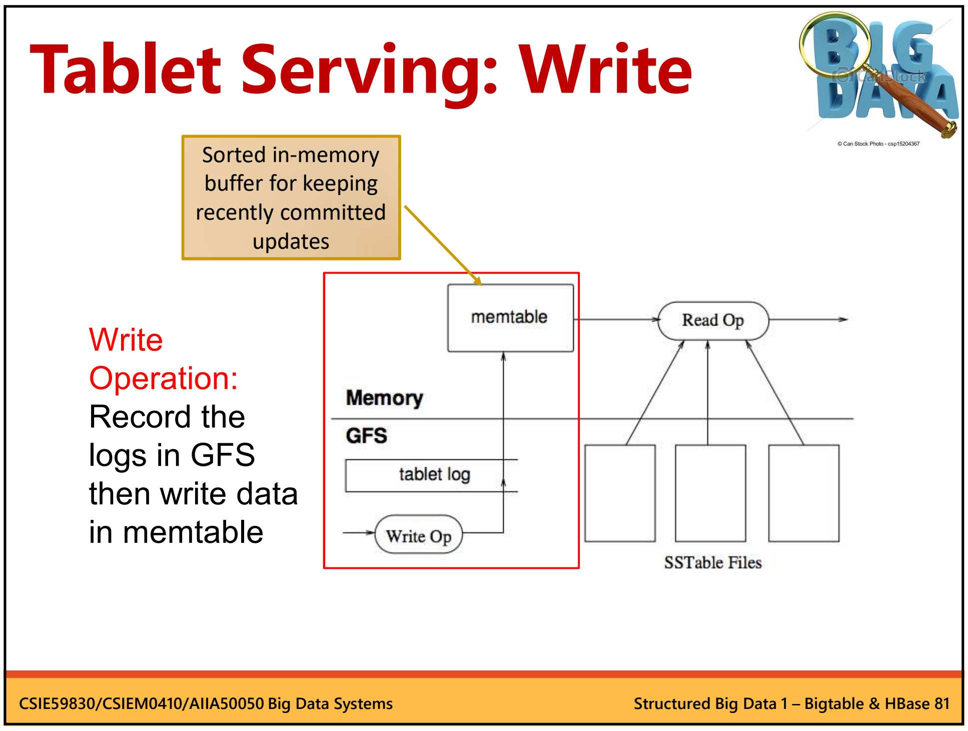
System Architecture: Locate Tablet


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



System Architecture : Serve Tablet

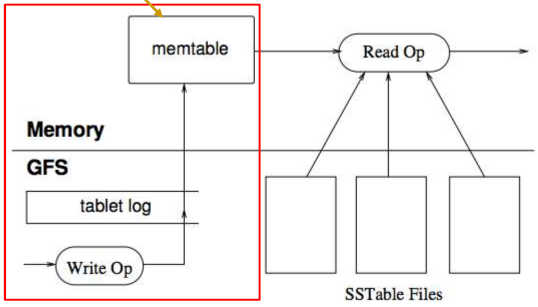

Bigtable Client
Bigtable Client Library
Bigtable Master
performs metadata ops, load balancing
Client fetch tablet from tablet server
Tablet Server
Serves Data
Tablet Server
Serves Data
Tablet Server
Serves Data
Cluster Scheduling System
handles failover, monitoring
GFS
holds tablet data, logs
Lock Service
holds metadata, handles master-election

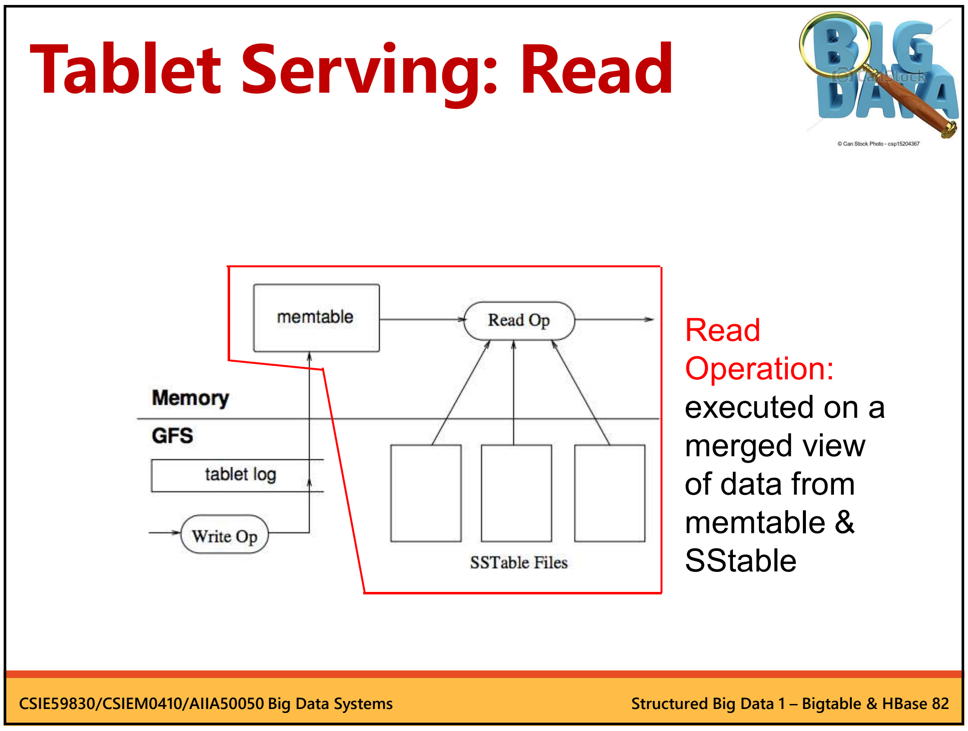


Tablet Serving: Write

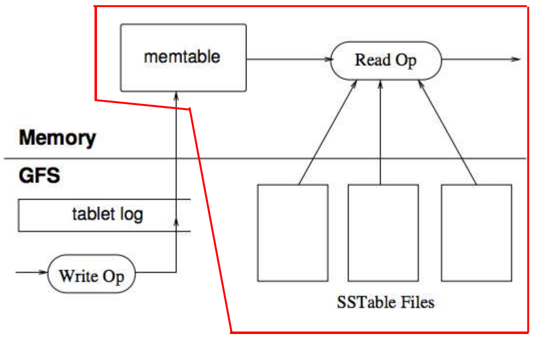
Sorted in-memory buffer for keeping recently committed updates

Write Operation: Record the logs in GFS then write data in memtable

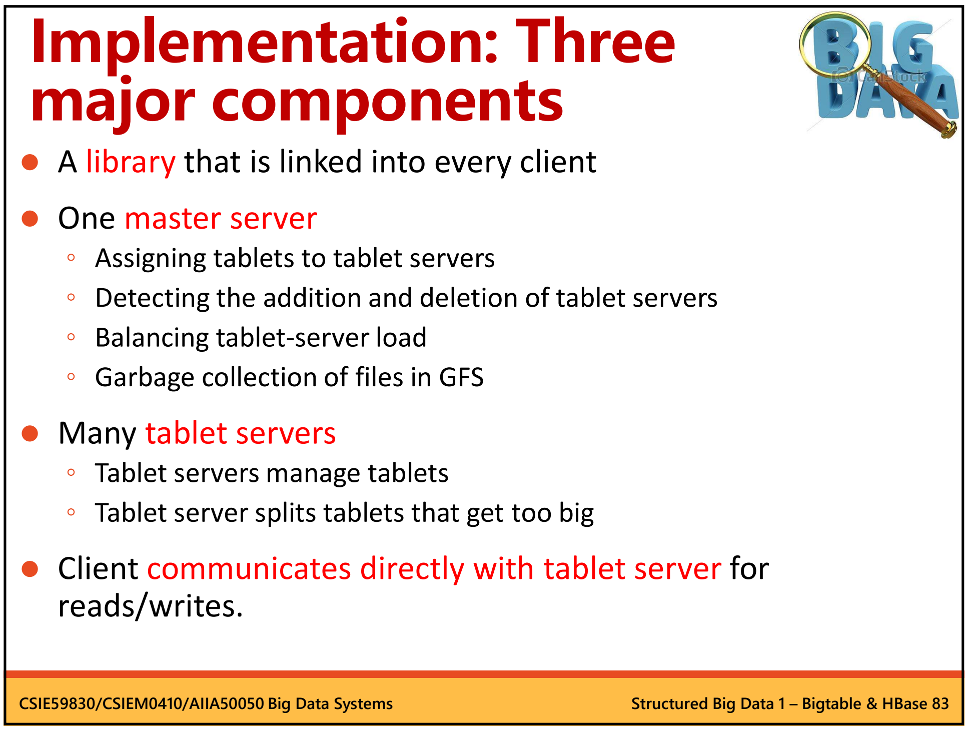




Tablet Serving: Read



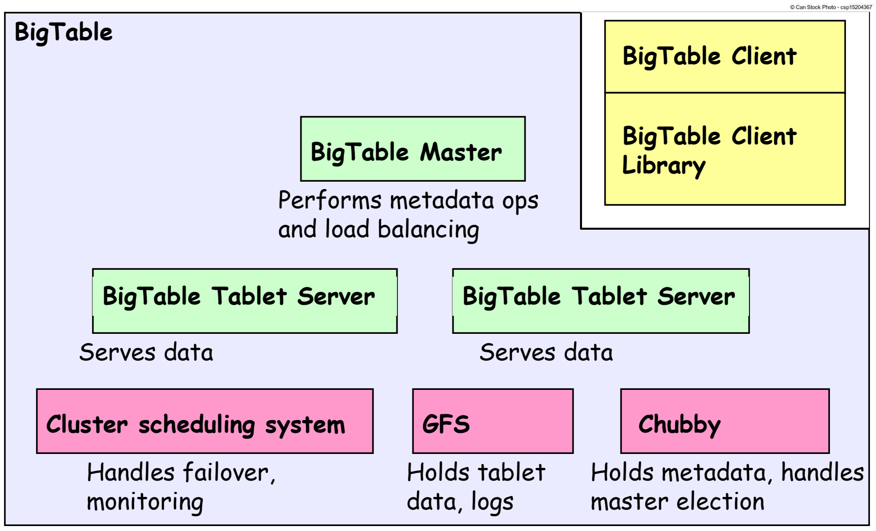
Read Operation: executed on a merged view of data from memtable & SStable

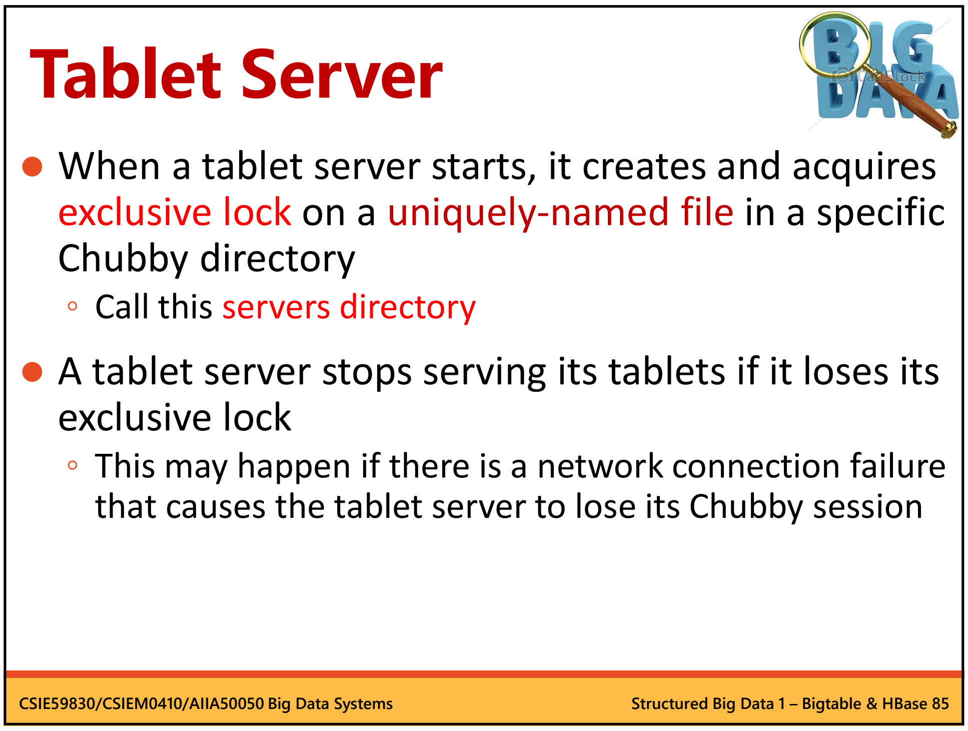


Implementation: Three major components

- 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.

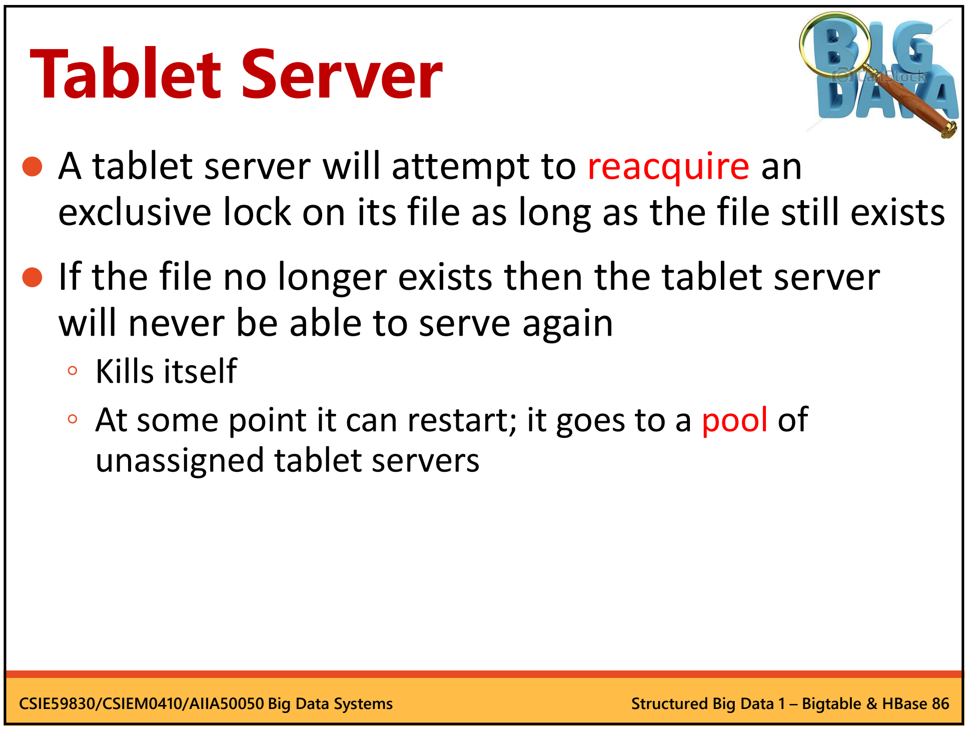
Architecture





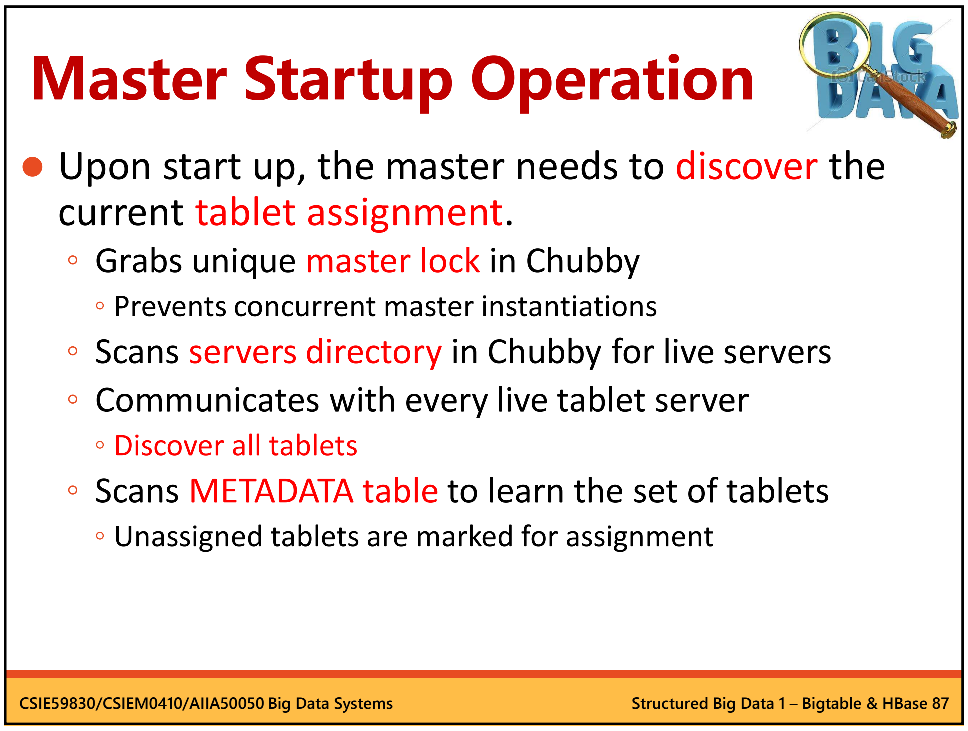
Tablet Server

- 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

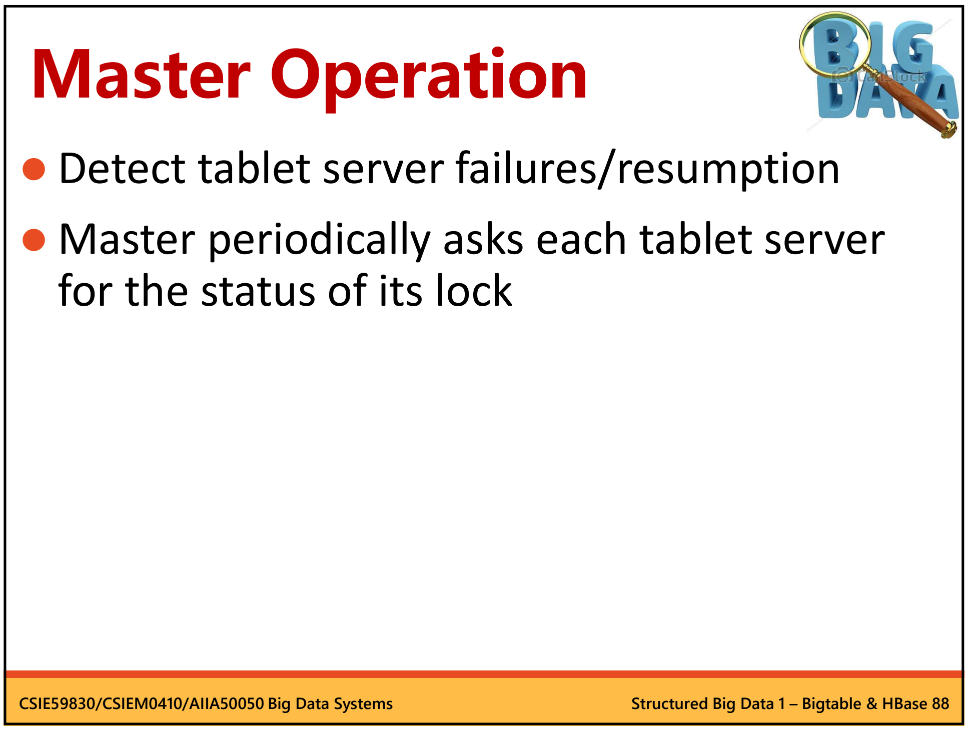
- A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists
- If the file no longer exists then the tablet server will never be able to serve again
  - Kills itself
  - At some point it can restart; it goes to a pool of unassigned tablet servers



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

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 87



Master Operation

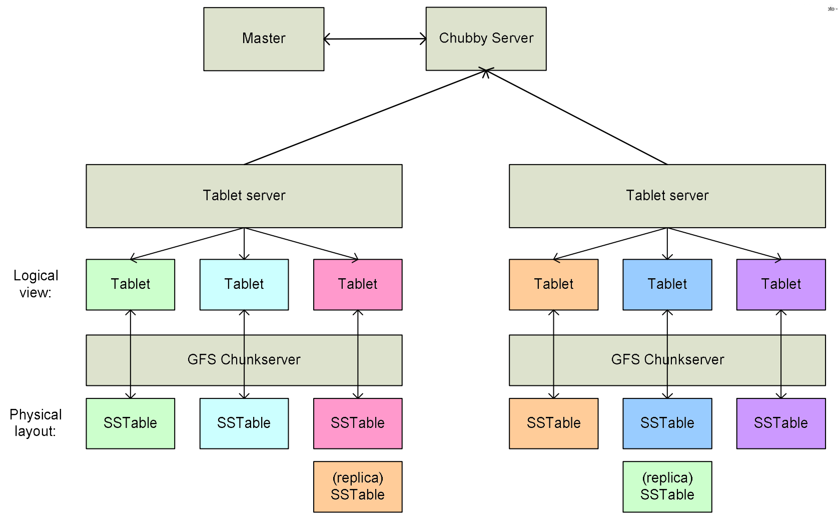
- Detect tablet server failures/resumption
- Master periodically asks each tablet server for the status of its lock

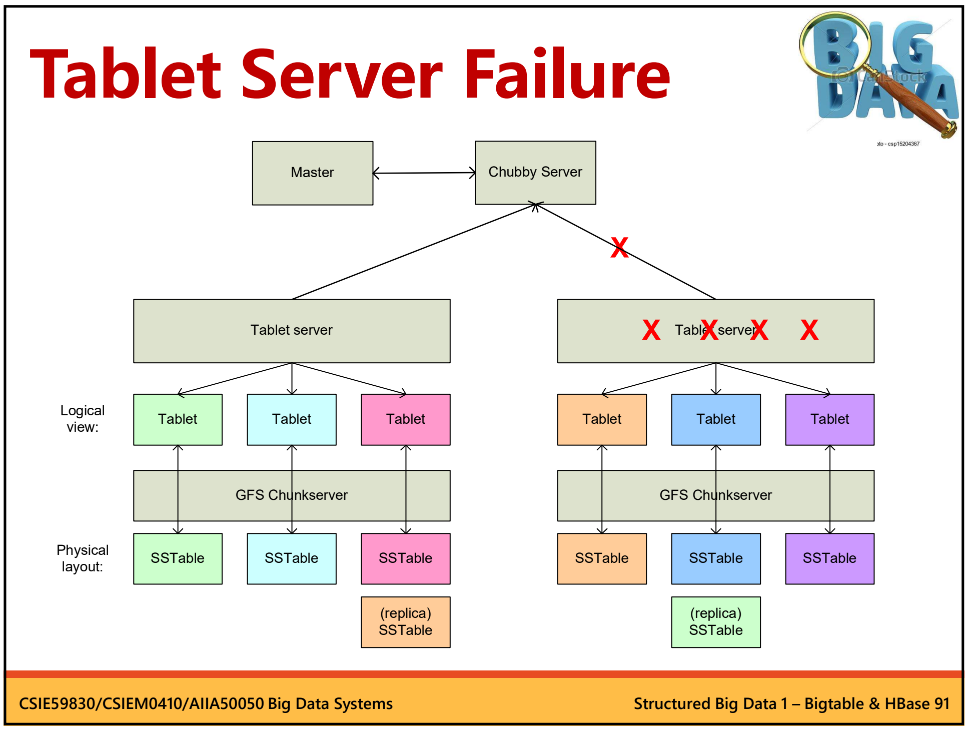
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems — Structured Big Data 1 – Bigtable & HBase 88

Master Operation

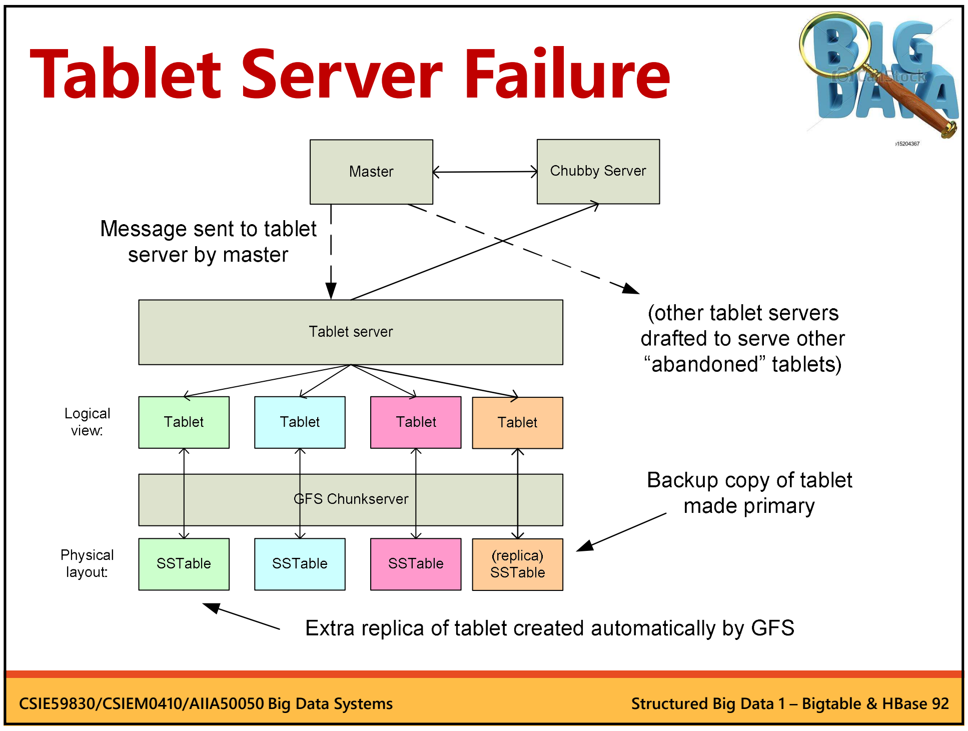
- Tablet server lost its lock or master cannot contact tablet server:
  - Master attempts to acquire exclusive lock on the server's file in the servers directory
  - If master acquires the lock then the tablets assigned to the tablet server are assigned to others
    - Master deletes the server's file 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



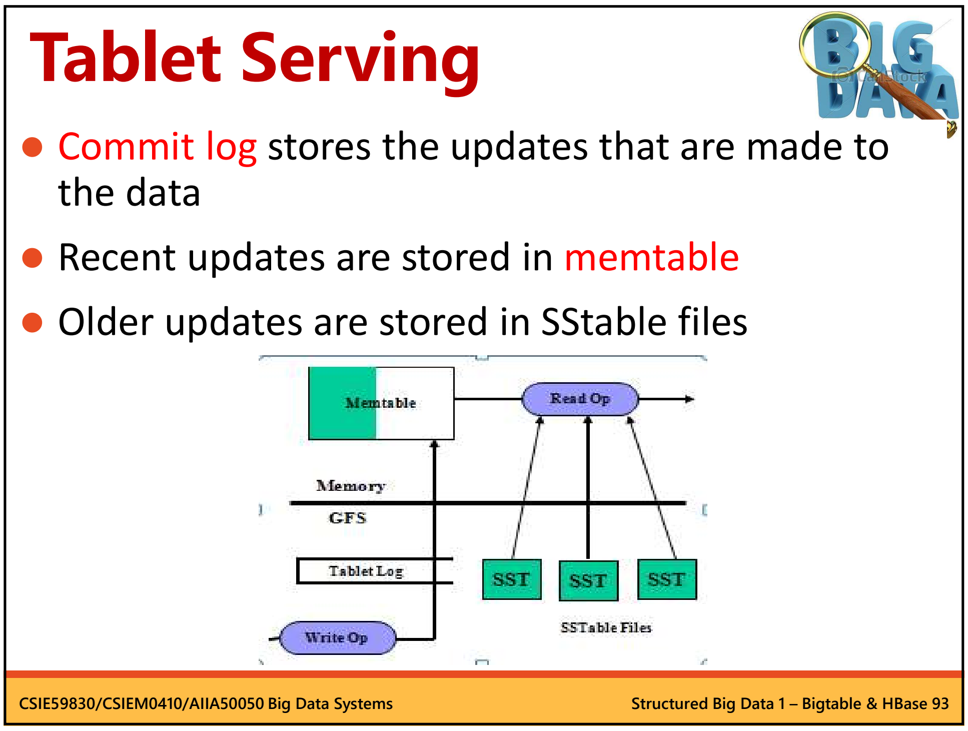


Tablet Server Failure



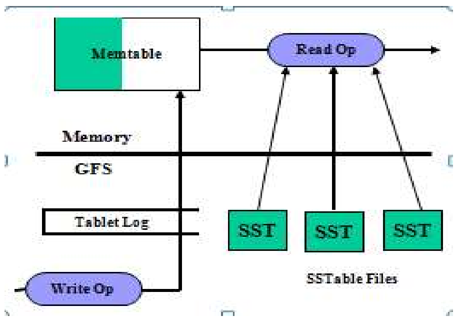
Tablet Server Failure

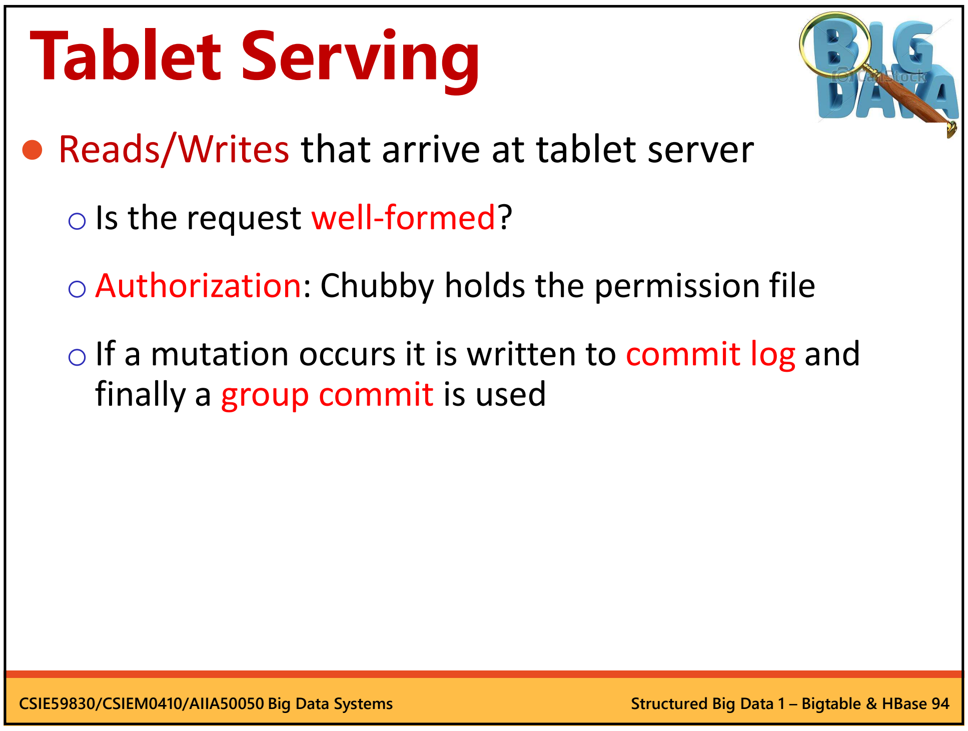
Master
Chubby Server
Message sent to tablet server by master
(other tablet servers drafted to serve other "abandoned" tablets)
Tablet server
Logical view:
Tablet
Tablet
Tablet
Tablet
GFS Chunkserver
Backup copy of tablet made primary
Physical layout:
SSTable
SSTable
SSTable
(replica) SSTable
Extra replica of tablet created automatically by GFS



Tablet Serving

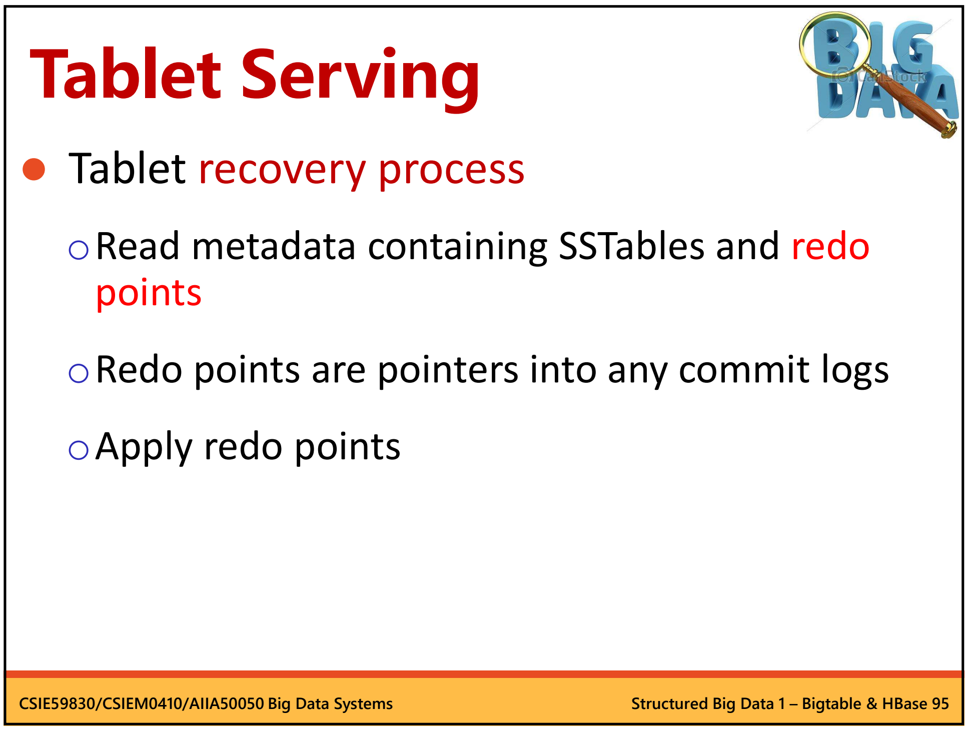
- Commit log stores the updates that are made to the data
- Recent updates are stored in memtable
- Older updates are stored in SStable files





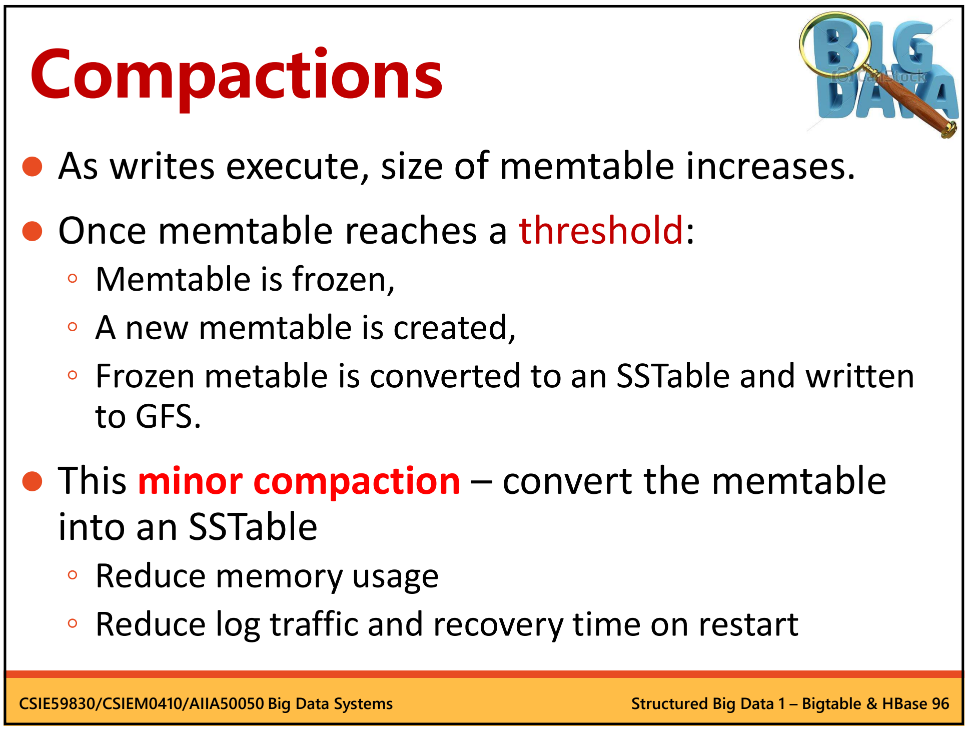
Tablet Serving

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



Tablet Serving

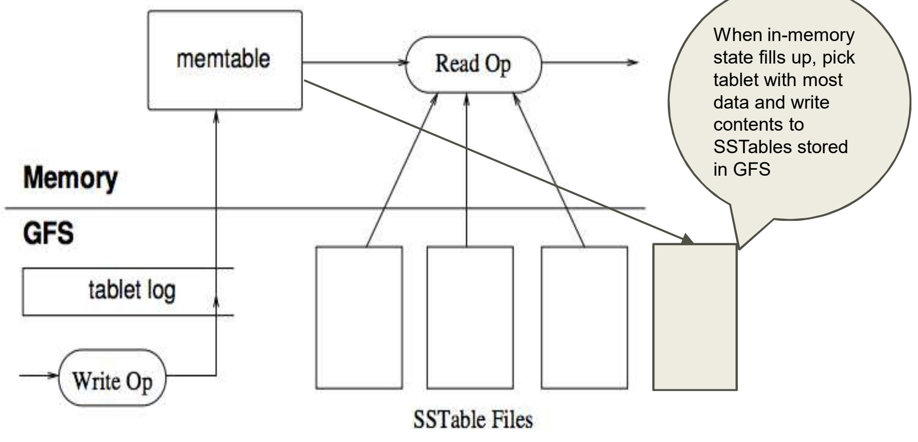
- Tablet recovery process
  - Read metadata containing SSTables and redo points
  - Redo points are pointers into any commit logs
  - Apply redo points



Compactions

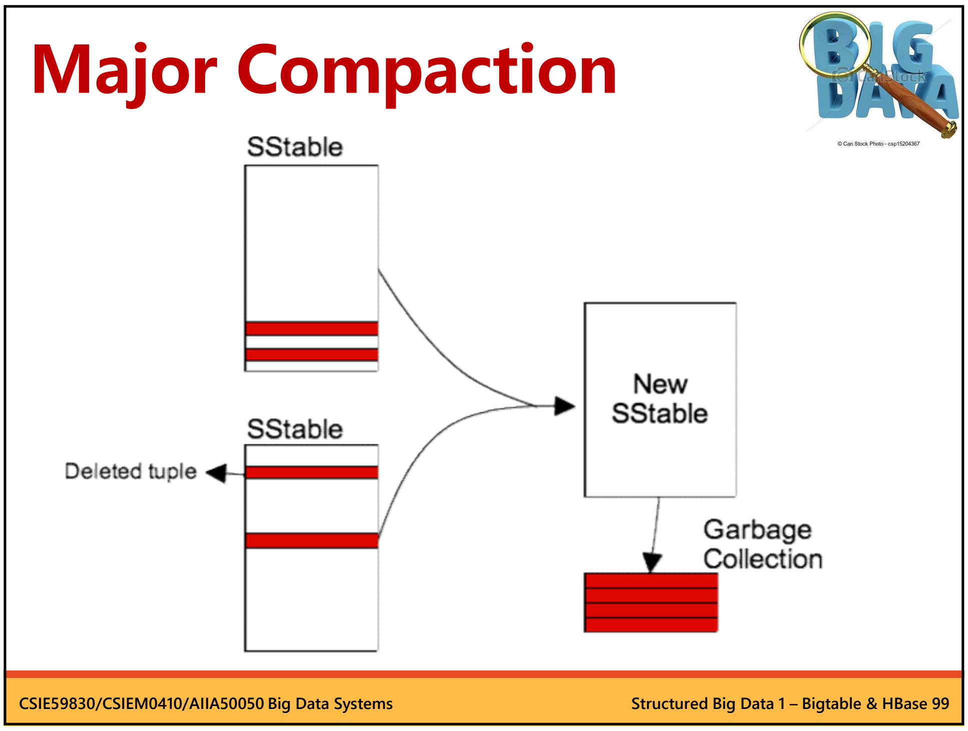
- As writes execute, size of memtable increases.
- Once memtable reaches a threshold:
  - Memtable is frozen,
  - A new memtable is created,
  - Frozen metable is converted to an SSTable and written to GFS.
- This **minor compaction** – convert the memtable into an SSTable
  - Reduce memory usage
  -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

Minor Compaction

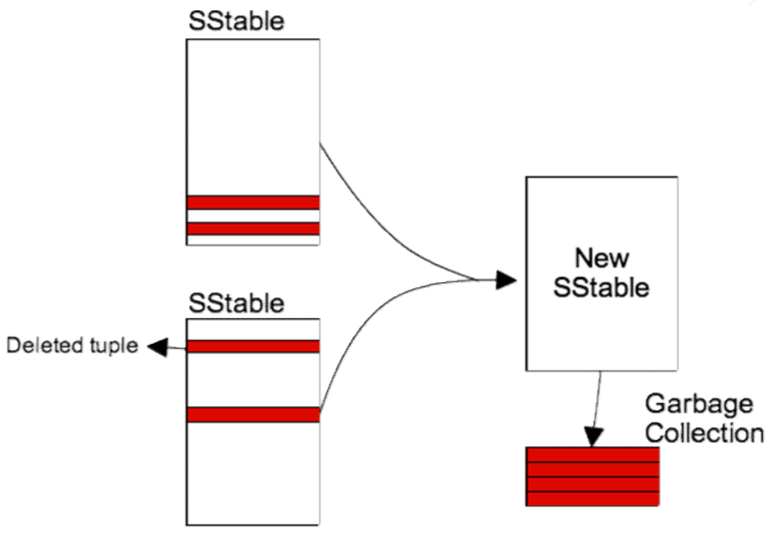

memtable
Read Op
Memory
GFS
tablet log
Write Op
SSTable Files
When in-memory state fills up, pick tablet with most data and write contents to SSTables stored in GFS

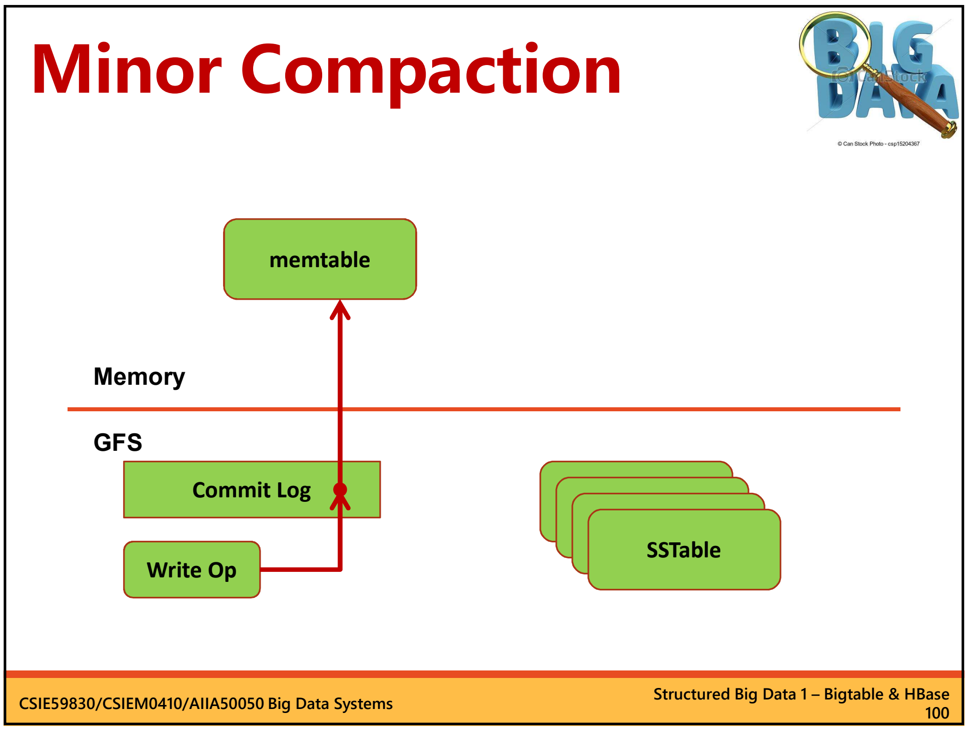
Compactions

- **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

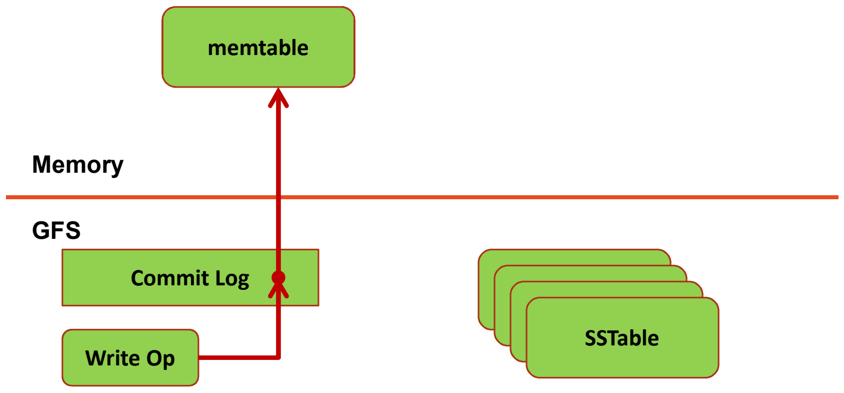


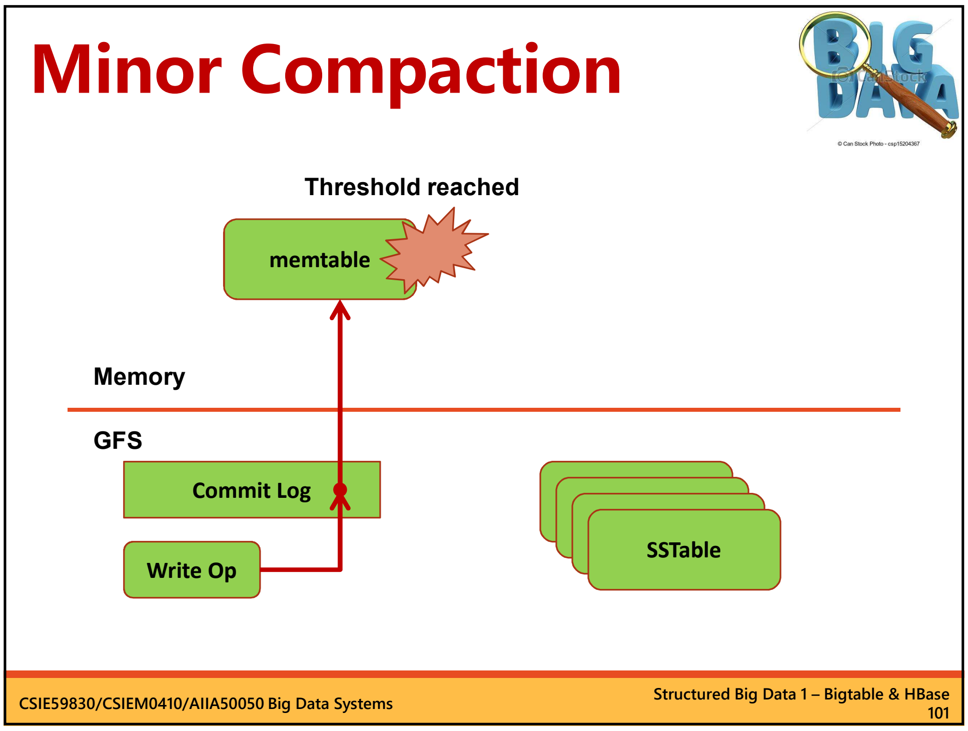
Major Compaction



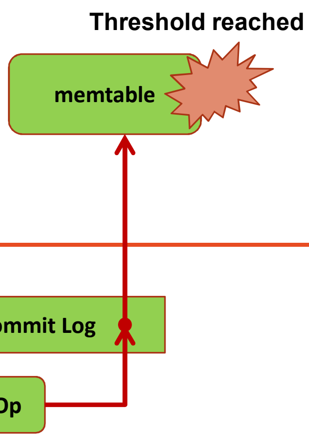


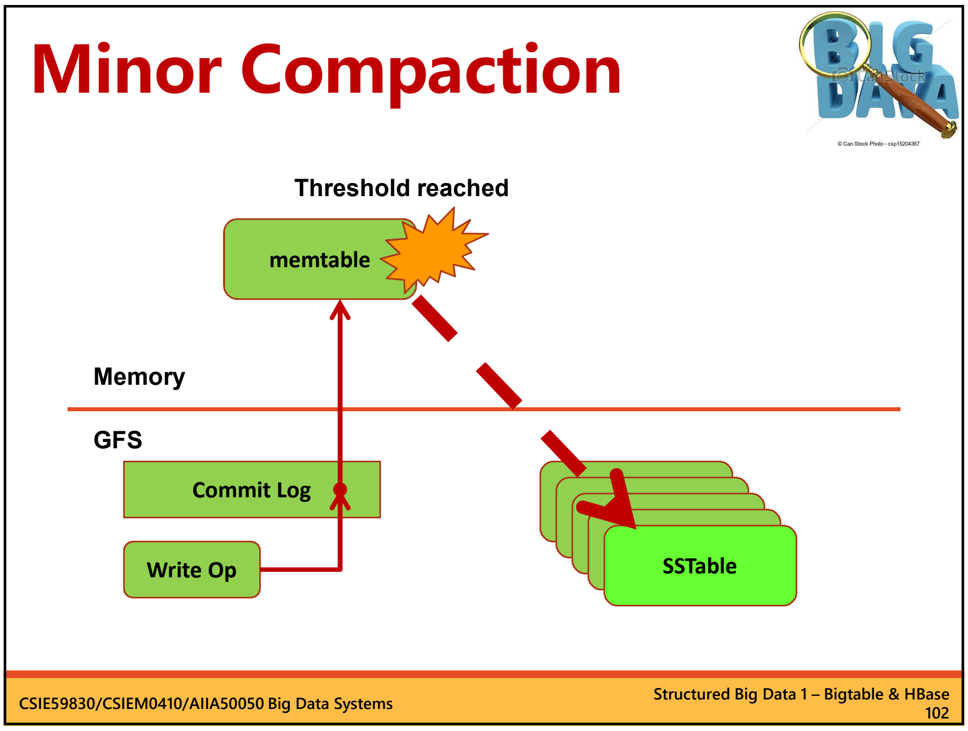
Minor Compaction



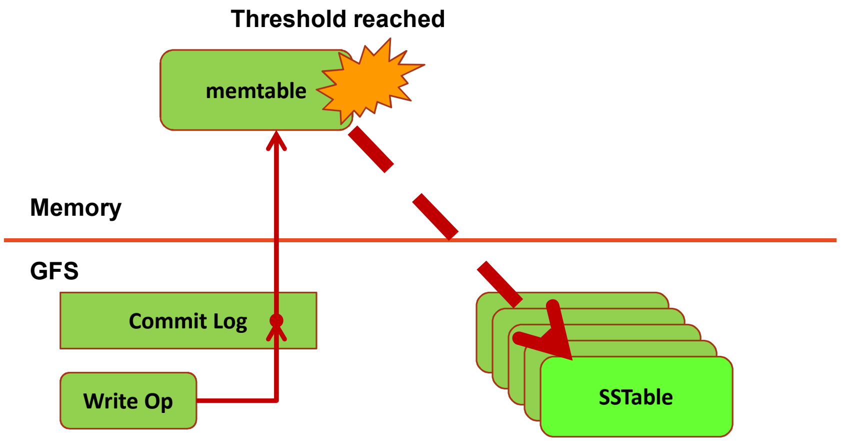


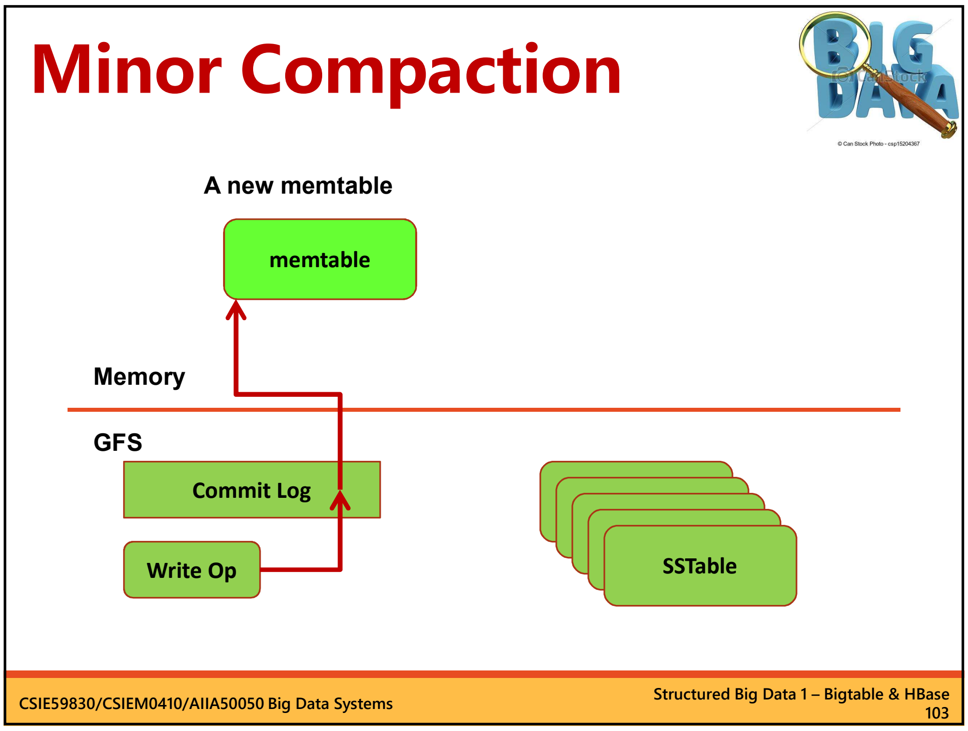
Minor Compaction



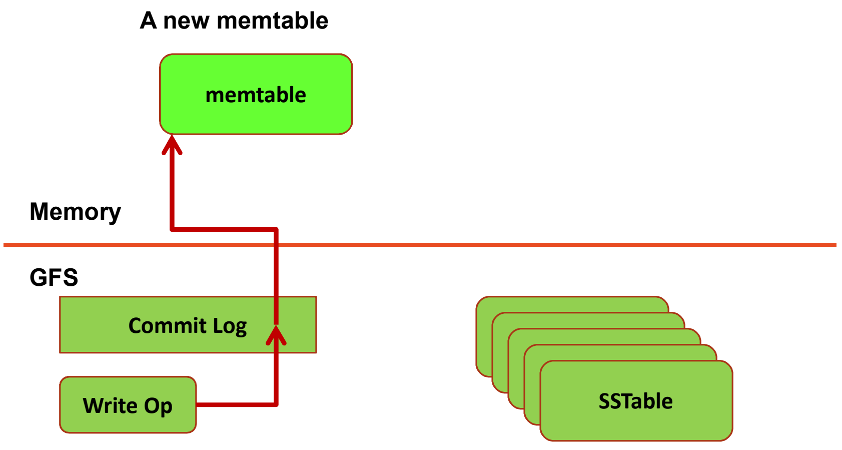


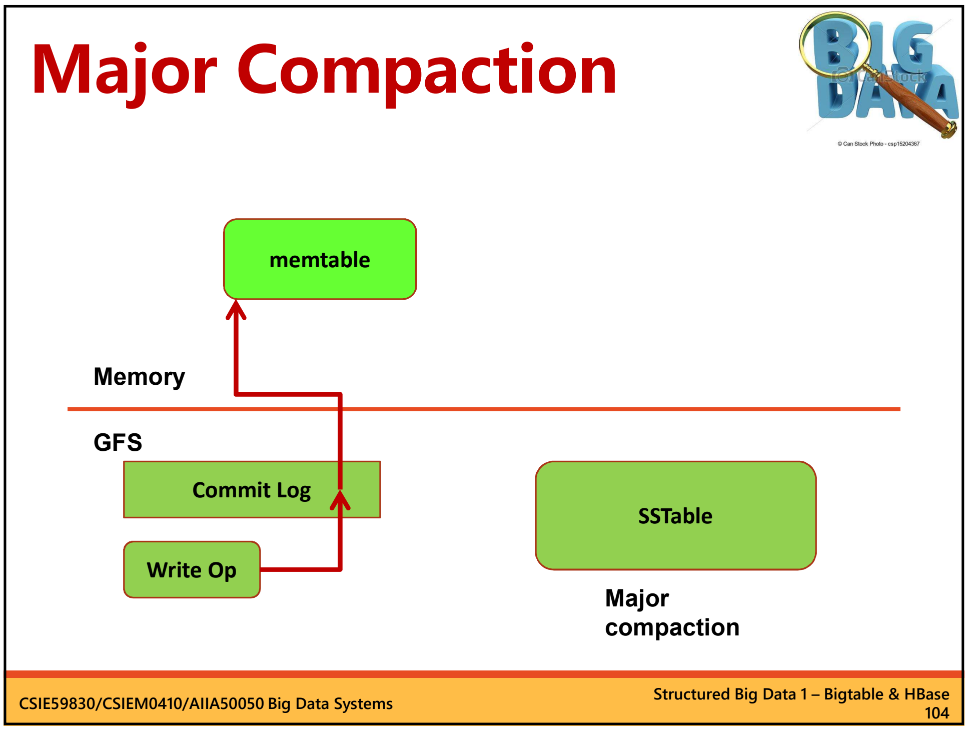
Minor Compaction



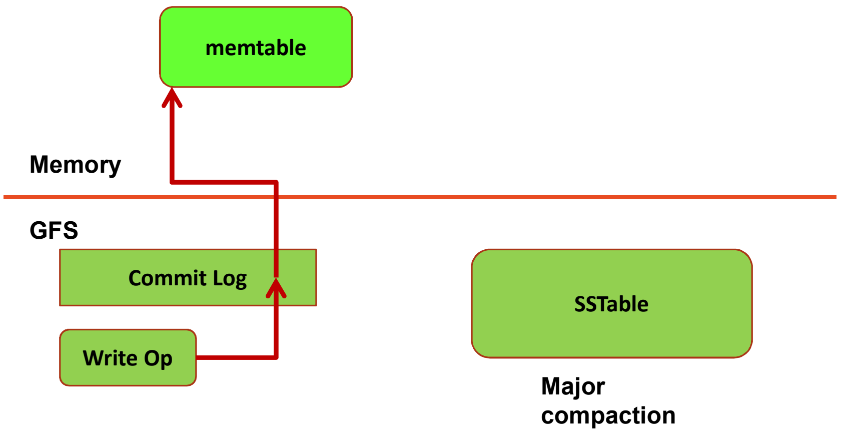


Minor Compaction





Major Compaction



System Performance

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

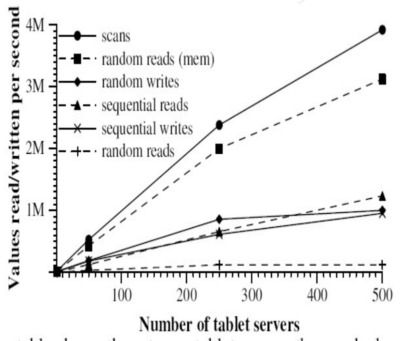
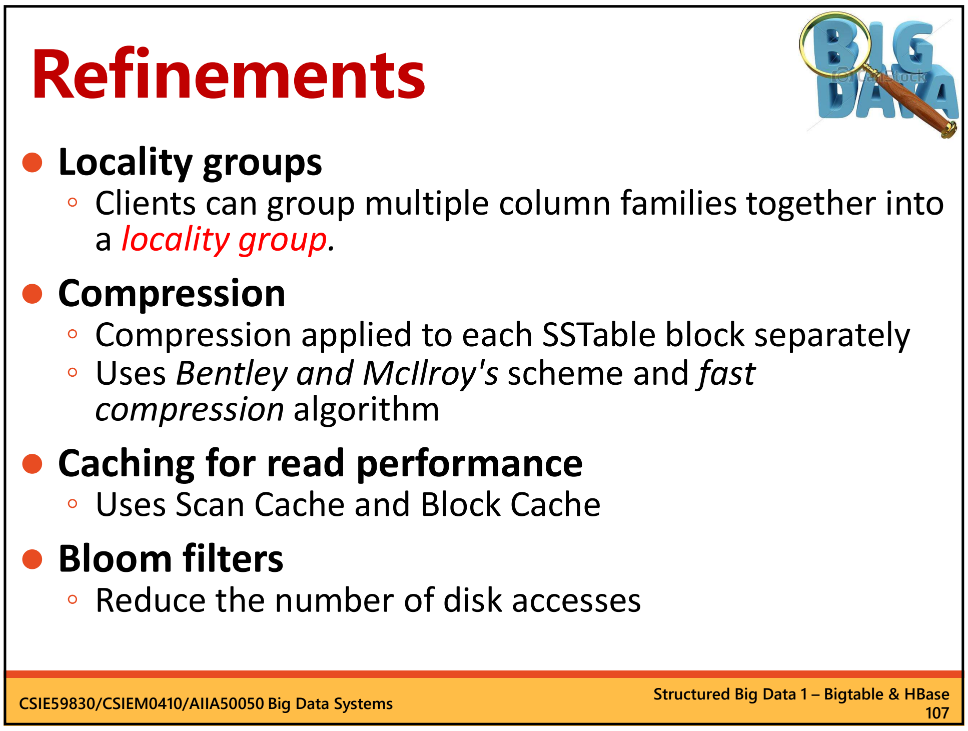


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

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

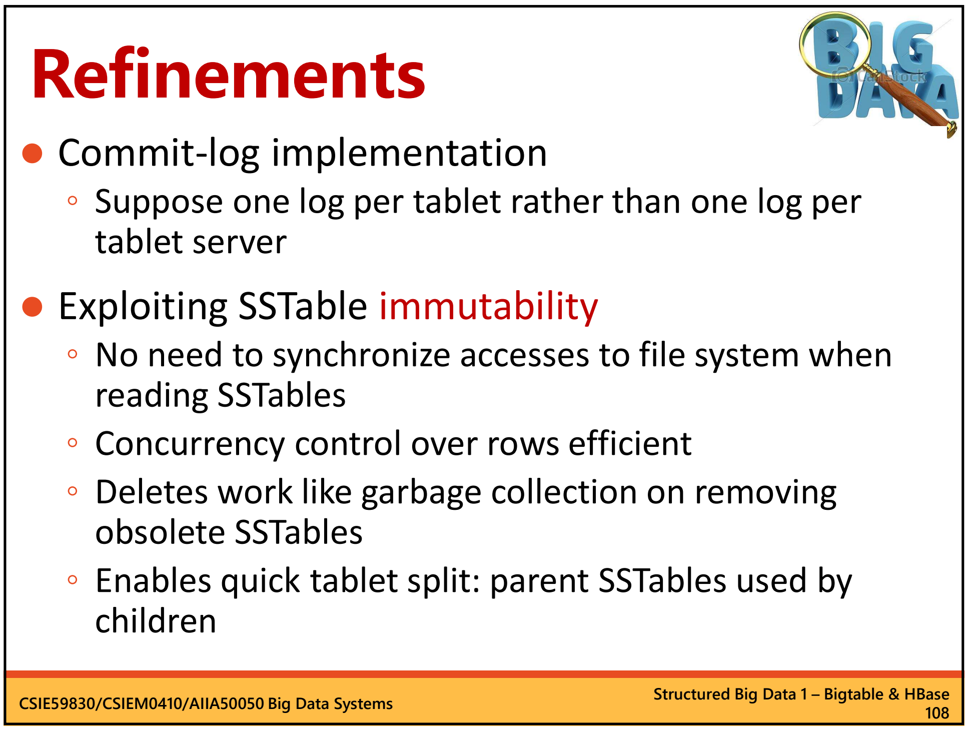
Performance Observation

- 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



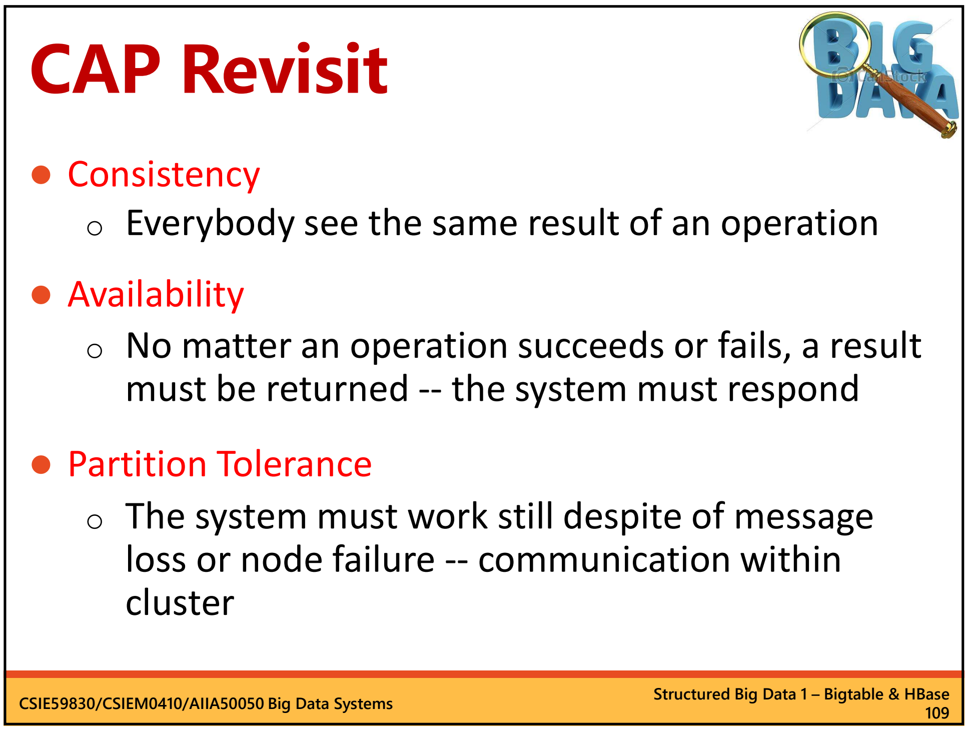
Refinements

- **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



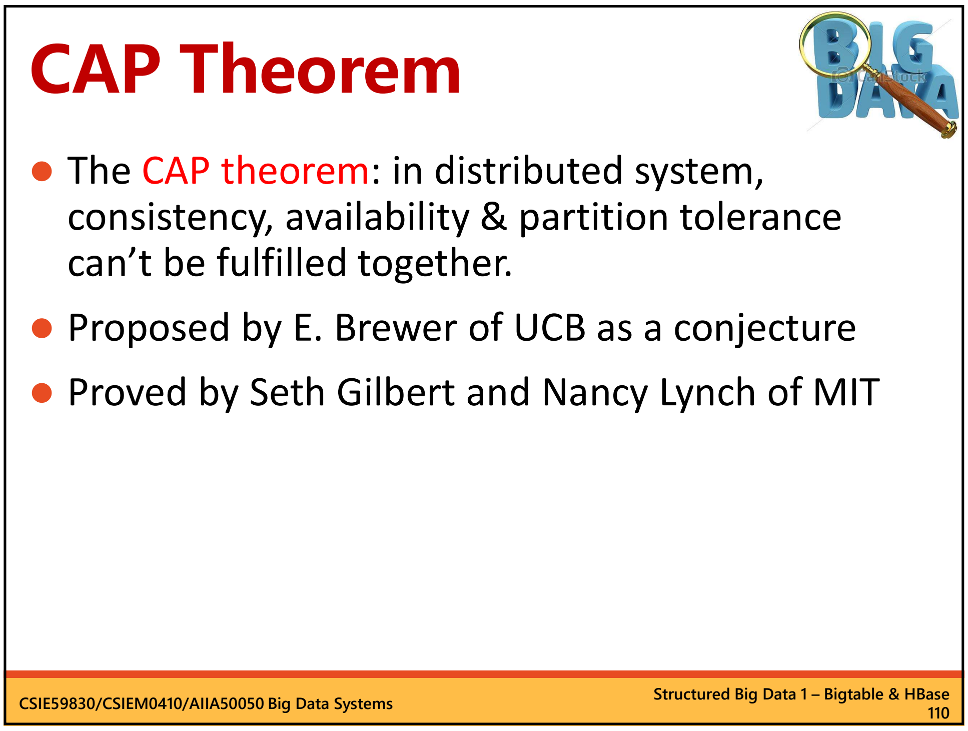
Refinements

- 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



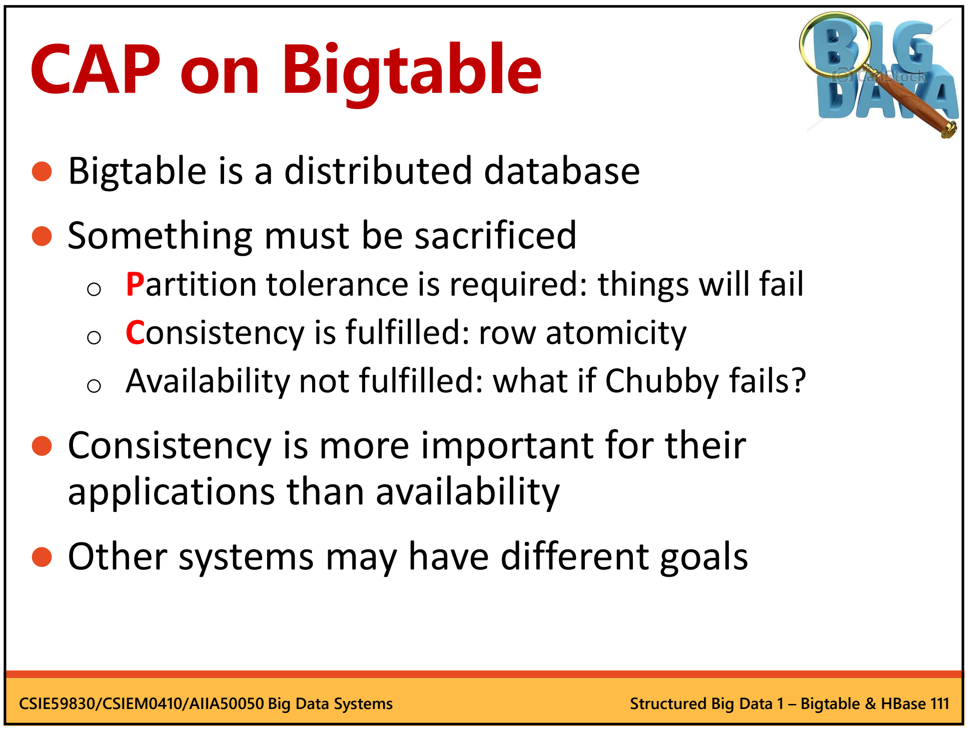
CAP Revisit

- 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



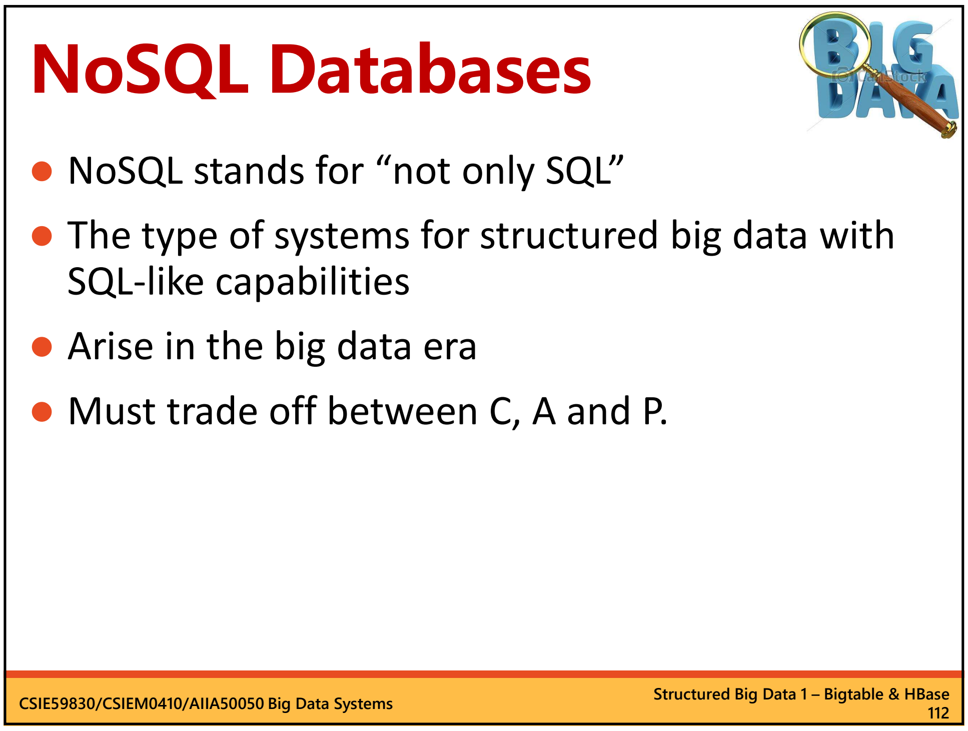
CAP Theorem

- 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



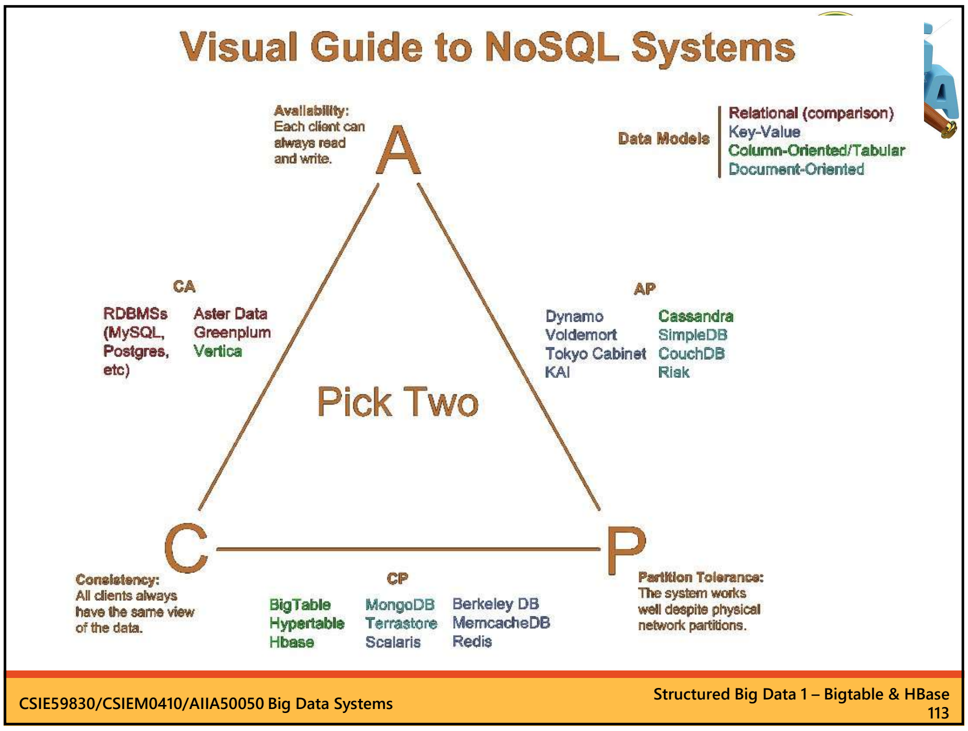
CAP on Bigtable

- 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

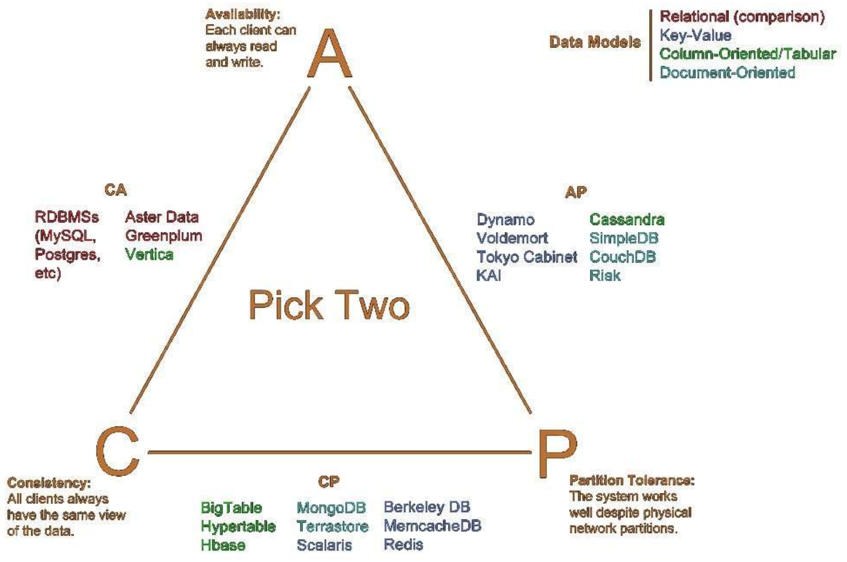


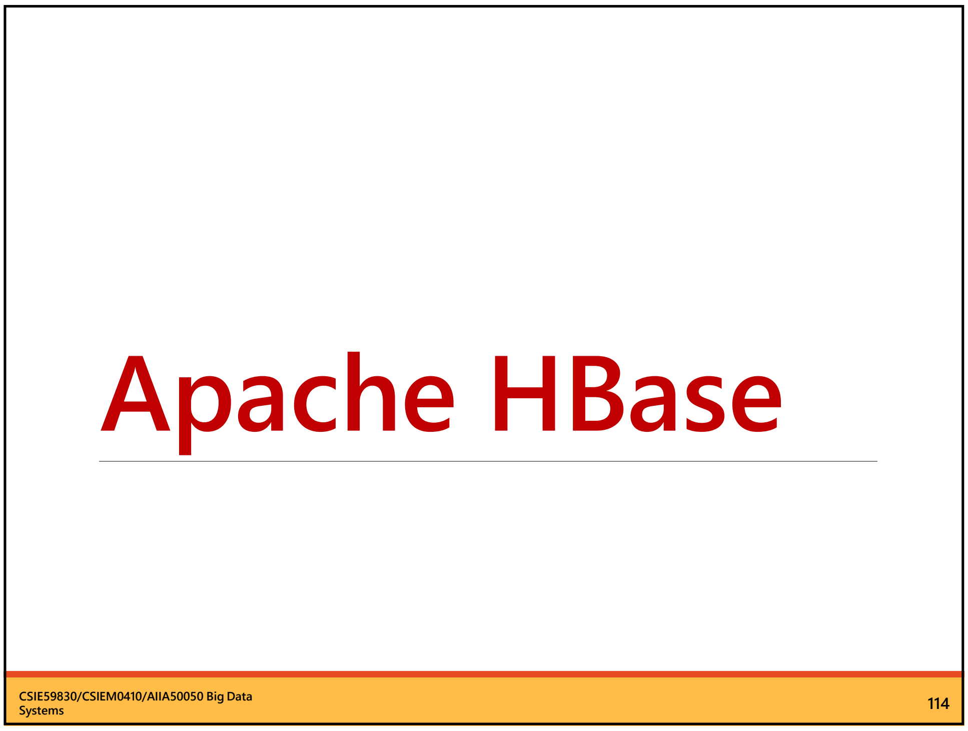
NoSQL Databases

- 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.

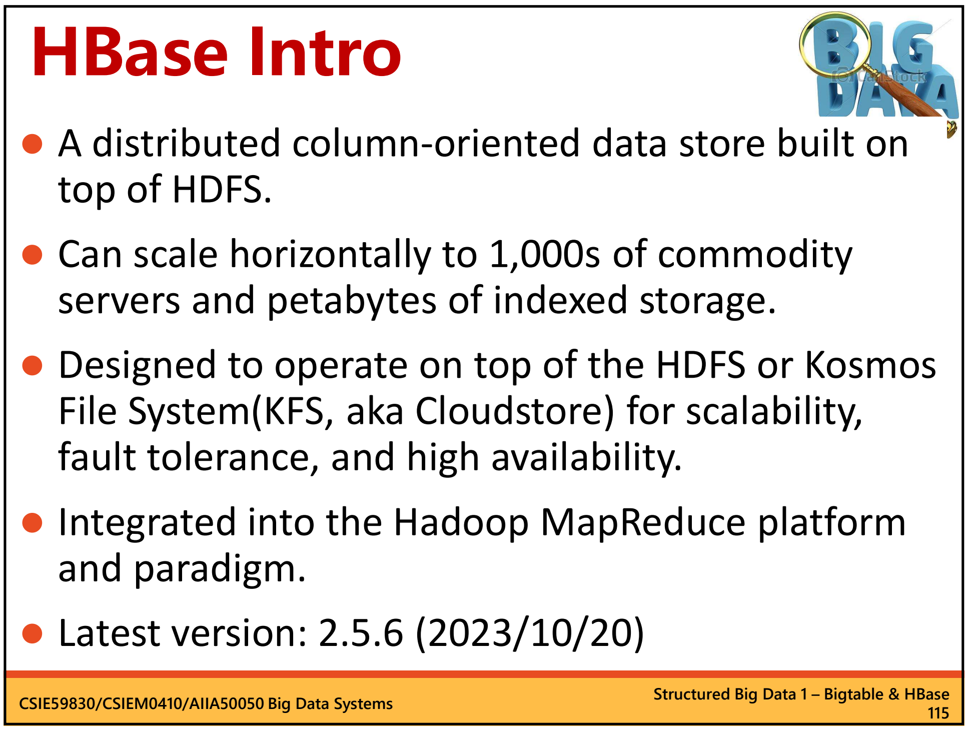


Visual Guide to NoSQL Systems





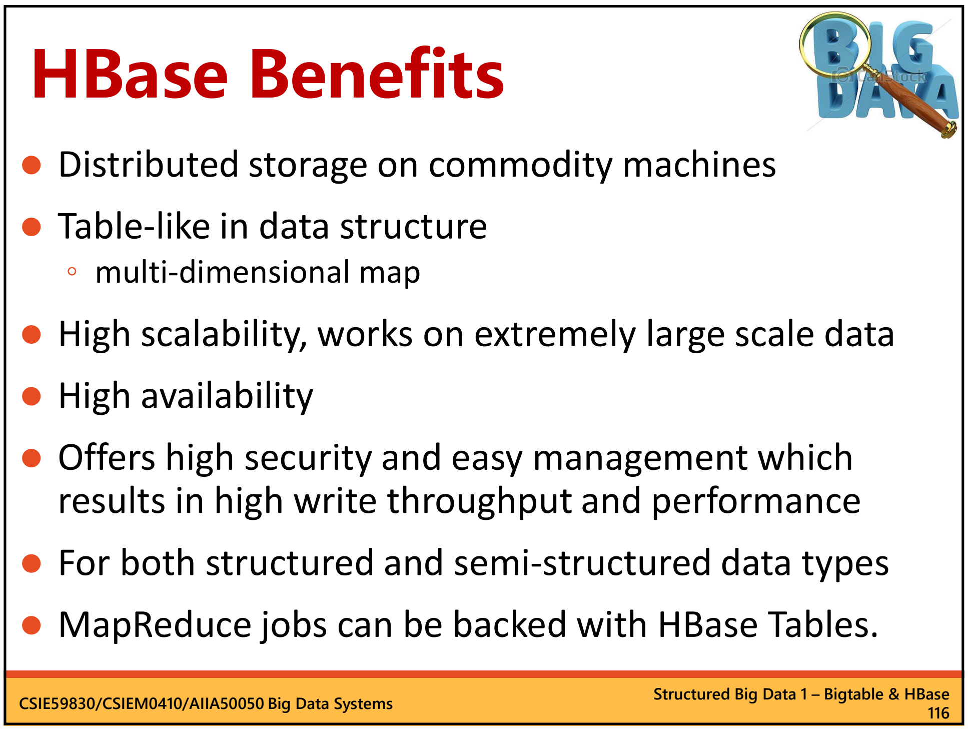
Apache HBase



HBase Intro

- A distributed column-oriented data store built on top of HDFS.
- Can scale horizontally to 1,000s of commodity servers and petabytes of indexed storage.
- Designed to operate on top of the HDFS or Kosmos File System(KFS, aka Cloudstore) for scalability, fault tolerance, and high availability.
- Integrated into the Hadoop MapReduce platform and paradigm.
- Latest version: 2.5.6 (2023/10/20)

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
Structured Big Data 1 – Bigtable & HBase
115



HBase Benefits

- Distributed storage on commodity machines
- Table-like in data structure
  - multi-dimensional map
- High scalability, works on extremely large scale data
- High availability
- Offers high security and easy management which results in high write throughput and performance
- For both structured and semi-structured data types
- MapReduce jobs can be backed with HBase Tables.

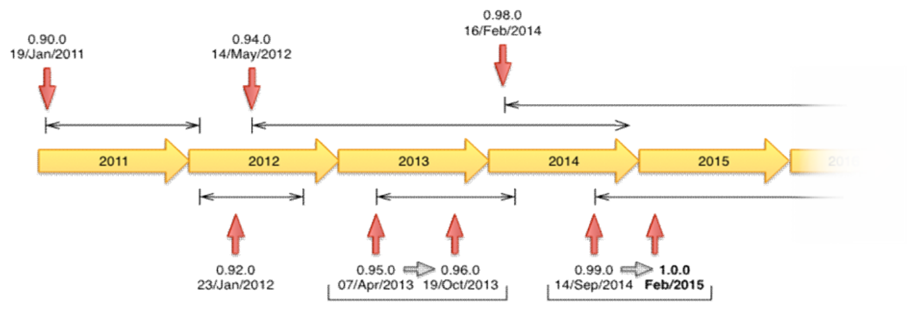
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
Structured Big Data 1 – Bigtable & HBase
116

HBase History

- Started by Chad Walters and Jim
- 2006.11
  - Google releases paper on BigTable
- 2007.2
  - Initial HBase prototype created as a Hadoop contribution.
- 2007.10
  - First useable HBase released (along with Hadoop 0.15.0)
- 2008.1
  - Hadoop became Apache top-level project and HBase became a subproject
- 2008.10~2009
  - HBase 0.18, 0.19, 0.20 released

HBase History

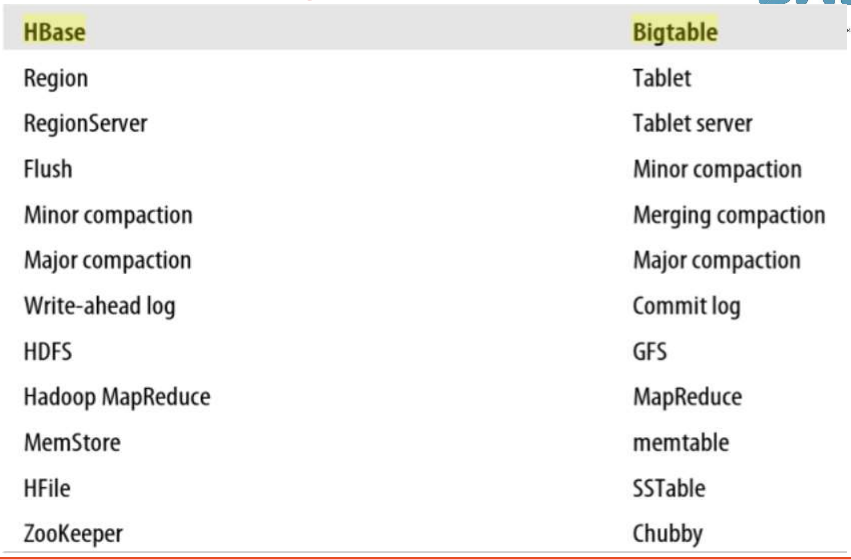
- 2010.05
  - HBase became an Apache top-level project
- 2010.06
  - HBase 0.89.20100621, first developer release



HBase History

- 2015~2017: HBase 1.1 ~ 1.4
- 2018.04.30: HBase 2.0.0, the second major release
- 2019.07~2022.08: HBase 2.2 ~ 2.5
- 2023.10.20: HBase 2.5.6 (latest release)
- 2021.07: HBase 3.0.0-alpha-1
- 2023.06: HBase 3.0.0-alpha-4
- Visit Apache HBase Project homepage (http://hbase.apache.org/) for details

HBase vs BigTable Terms



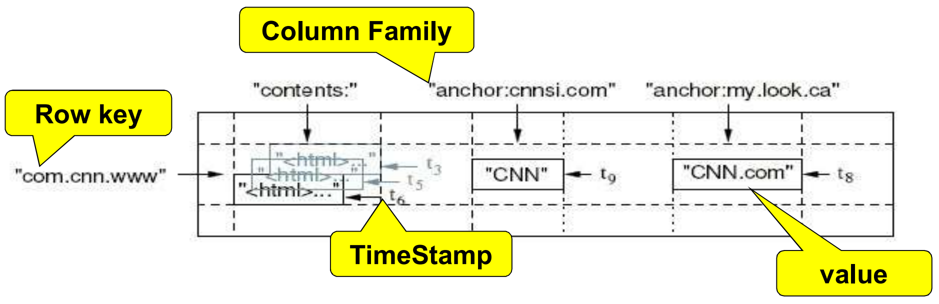
HBase	Bigtable
Region	Tablet
RegionServer	Tablet server
Flush	Minor compaction
Minor compaction	Merging compaction
Major compaction	Major compaction
Write-ahead log	Commit log
HDFS	GFS
Hadoop MapReduce	MapReduce
MemStore	memtable
HFile	SSTable
ZooKeeper	Chubby

Data Model

- Almost the same as BigTable
- Tables are sorted by Row Key
- Table schema only define it's *column families* .
  - Each family consists of any number of columns
  - Each column consists of any number of versions
  - Columns only exist when inserted, NULLs are free.
  - Columns within a family are sorted and stored together
- Everything except table names are byte[]
- (Row, Family:Column, Timestamp) → Value

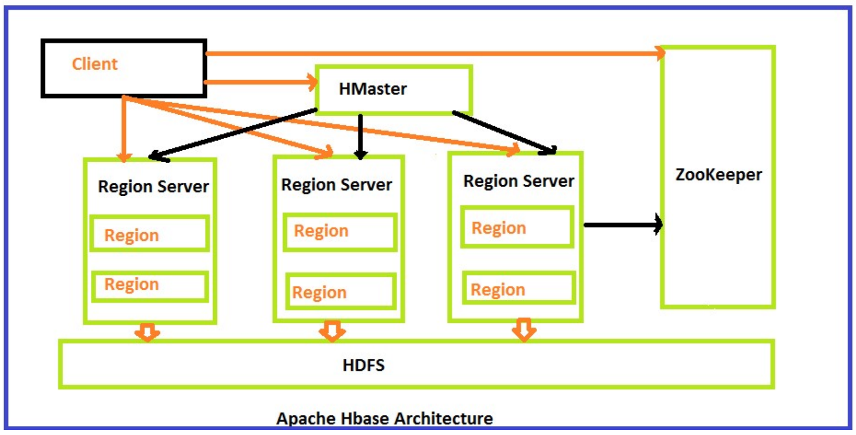
Data Model

- Very similar to BigTable



HBase Architecture

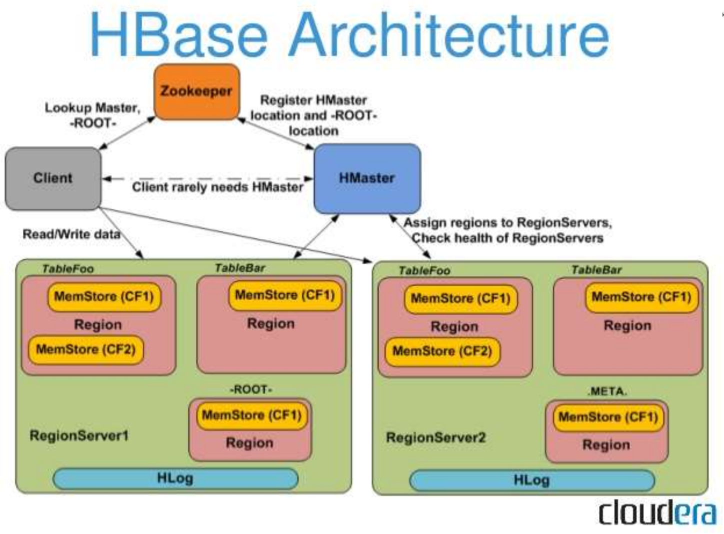
- A simplified architecture of HBase (details later)



Components

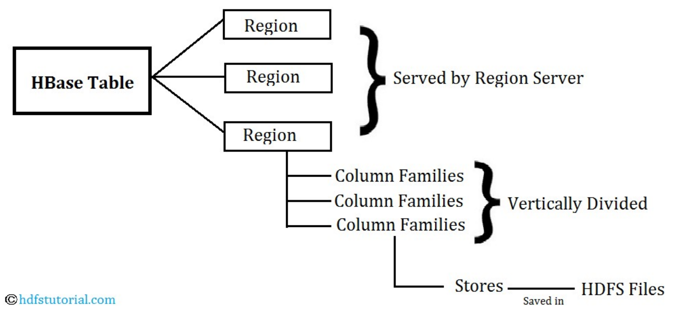
- Zookeeper
  - Distributed coordination service
  - Maintain cluster status and server failure notification
- Master Server (HMaster)
  - Responsible for monitoring region servers
  - Load balancing for regions
  - Redirect client to correct region servers
  - The current SPOF
- Region Servers (slaves)
  - Serving requests(Write/Read/Scan) of Client
  - Send HeartBeat to Master
  - Throughput and Region numbers are scalable by region servers

HBase Architecture



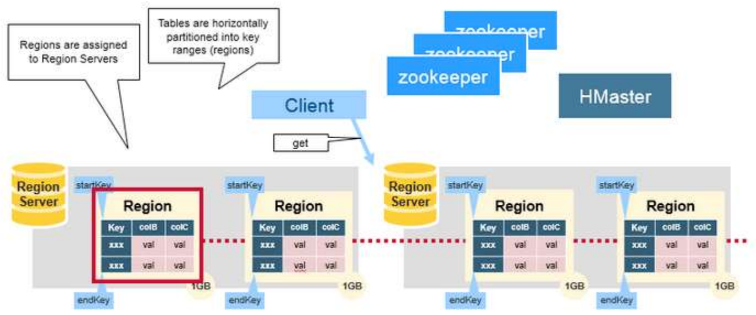
Tables & Regions

- HBase tables are divided into regions and are being served by region servers.
- Regions are divided into Column Families vertically into Stores. And then stores are saved in HDFS files.



Regions

- Tables are divided horizontally by row-key range (with starKey and endKey) into regions.
- Regions are assigned to region servers. A single region server can server around 1000 regions.



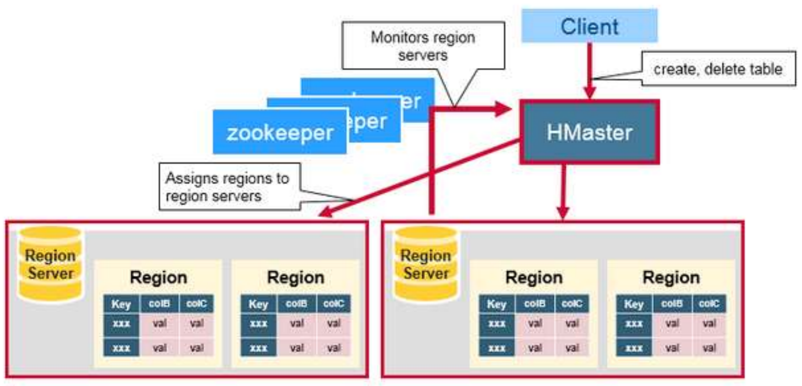
Region Servers

- Manages regions and runs on HDFS DataNodes.
- When tables grow beyond the configurable limit, HBase system automatically splits the table and distributes the load to another Region Server.
- The process is called auto-sharding.
- Communicates with the client and handles data-related operations
- Decide the size of the region
- Handle the read and write requests for all the regions under it.

HMaster Server

- Assigns regions to region server with the help of Zookeeper
- Responsible for load balancing. Unload busy servers and assign regions to less occupied servers.
- Responsible for schema changes like HBase table creation, the creation of column families etc.
- Interface for creating, deleting, updating tables
- Monitor all the region servers in the cluster

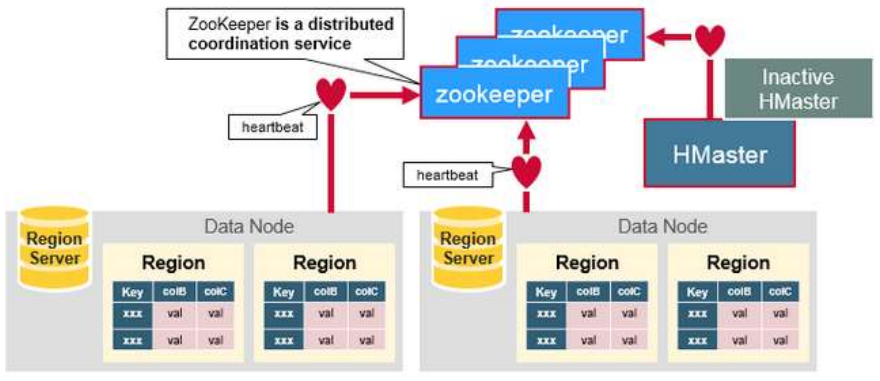
HMaster Server



ZooKeeper: The Coordinator

- HBase uses ZooKeeper as a distributed coordination service to maintain server state.
- Zookeeper maintains which servers are alive and available, and provides server failure notification.
- Zookeeper uses consensus to guarantee common shared state. (3 or 5 machines for consensus)
- Provides distributed synchronization
- Client communication establishment with region servers
- Master and region servers registered themselves with ZooKeeper(ZK). The client needs access to ZK quorum configuration to connect with master and region servers.

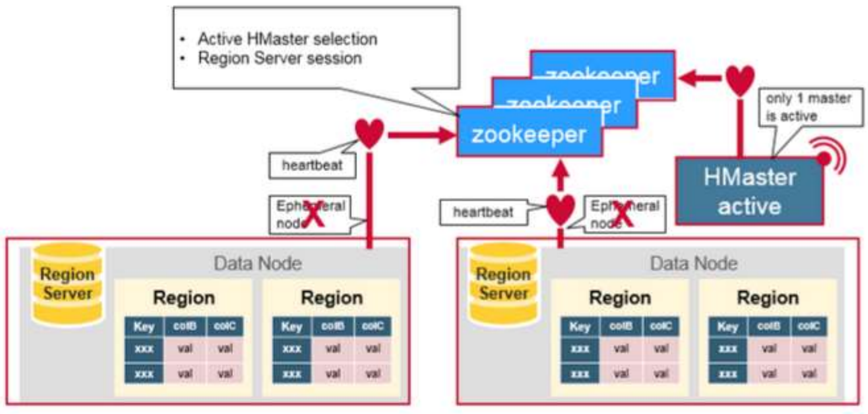
ZooKeeper: The Coordinator

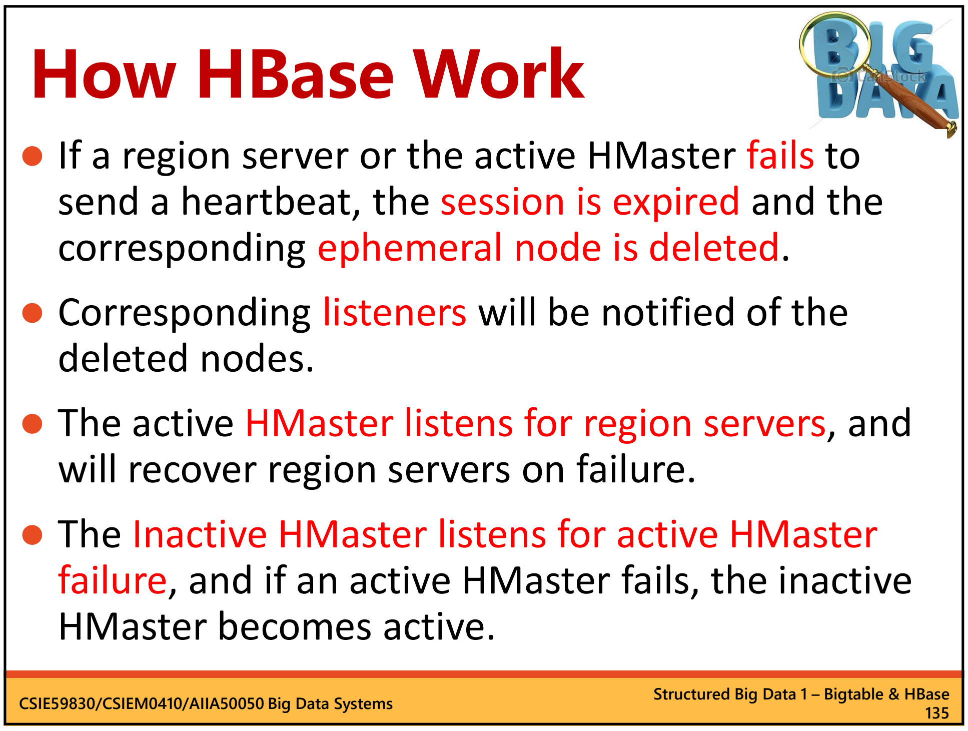


How HBase Work

- Zookeeper is used to coordinate shared state info.
- Region servers and the active HMaster connect with a session to ZooKeeper.
- ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.
- Each Region Server creates an ephemeral node.
- HMaster monitors these nodes to discover available region servers and server failures.
- Zookeeper makes sure that only one master is active.
- The active HMaster sends heartbeats to Zookeeper.
- The inactive HMaster listens for notifications of the active HMaster failure.

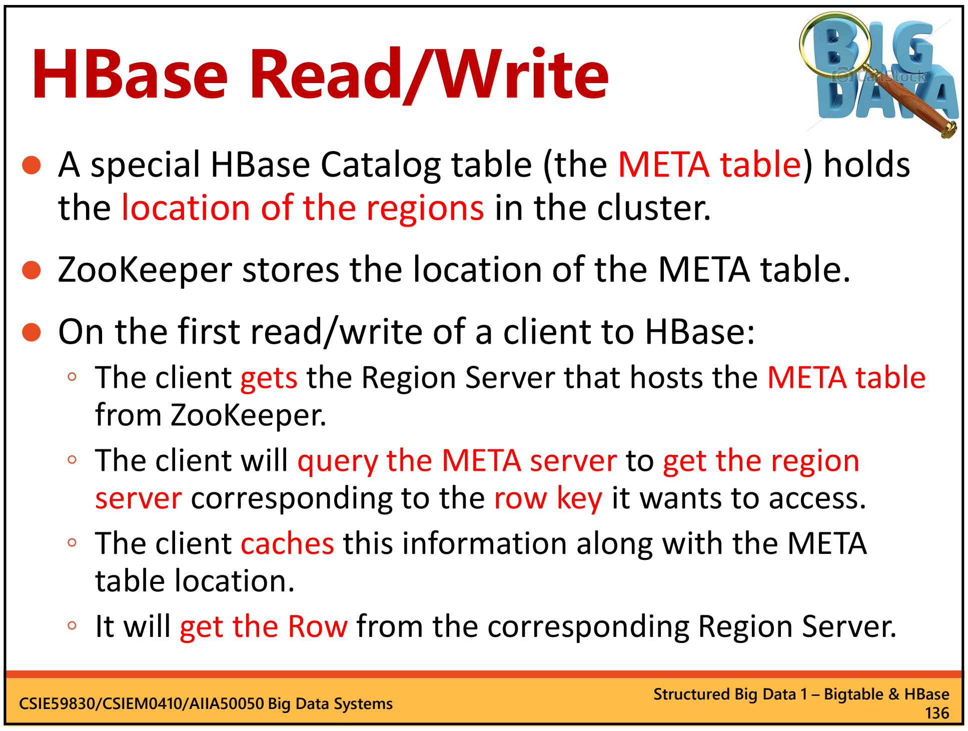
How HBase Work





How HBase Work

- If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted.
- Corresponding listeners will be notified of the deleted nodes.
- The active HMaster listens for region servers, and will recover region servers on failure.
- The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

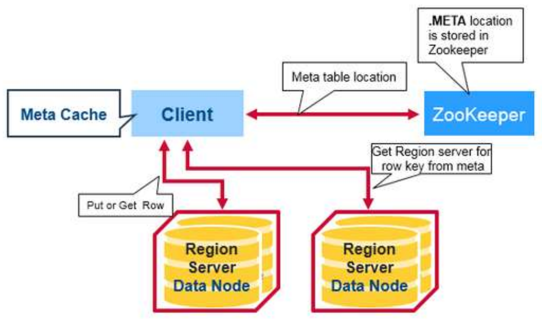


HBase Read/Write

- A special HBase Catalog table (the META table) holds the location of the regions in the cluster.
- ZooKeeper stores the location of the META table.
- On the first read/write of a client to HBase:
  - The client gets the Region Server that hosts the META table from ZooKeeper.
  - The client will query the META server to get the region server corresponding to the row key it wants to access.
  - The client caches this information along with the META table location.
  - It will get the Row from the corresponding Region Server.

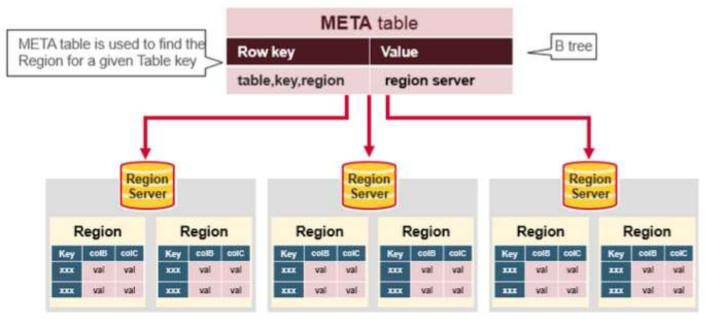
HBase Read/Write

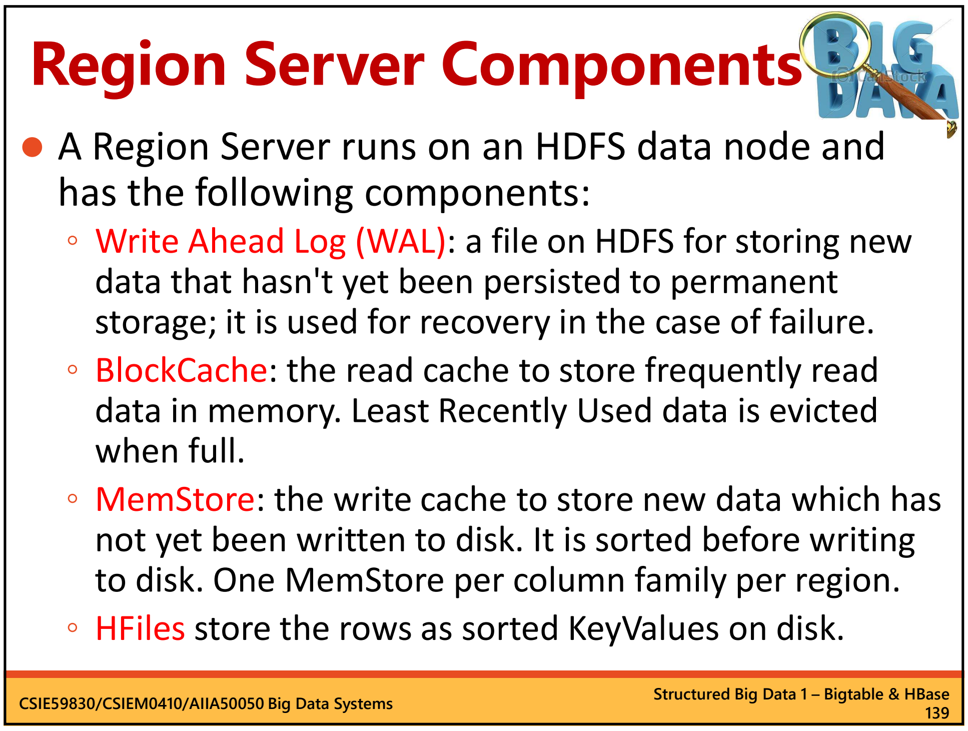
- For more reads, the client uses the cache for the META location and previously read row keys.
- It does not need to query the META table unless there is a miss (region has moved), then it will re-query and update the cache.



HBase Meta Table

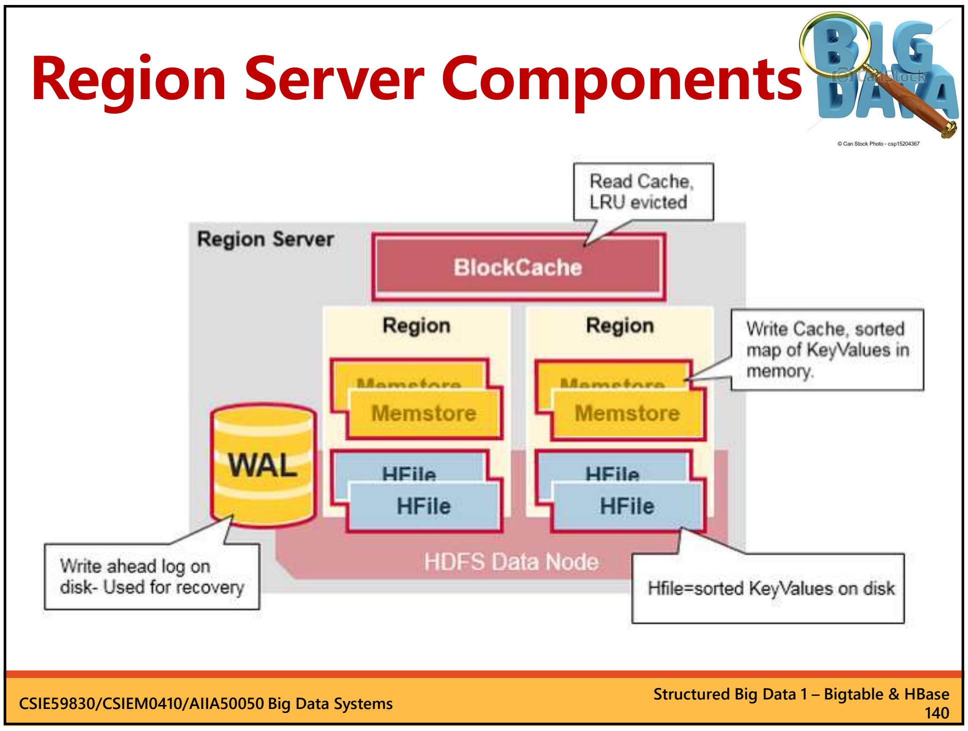
- META table keeps a list of all regions in the system.
- META table is like a B tree and is structured as:
  - Key: region start key, region id
  - Values: Region Server



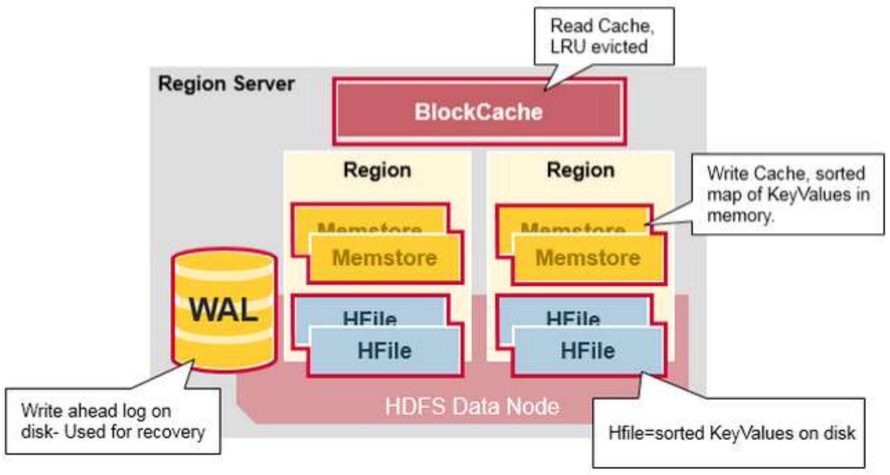


Region Server Components

- A Region Server runs on an HDFS data node and has the following components:
  - Write Ahead Log (WAL): a file on HDFS for storing new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
  - BlockCache: the read cache to store frequently read data in memory. Least Recently Used data is evicted when full.
  - MemStore: the write cache to store new data which has not yet been written to disk. It is sorted before writing to disk. One MemStore per column family per region.
  - HFiles store the rows as sorted KeyValues on disk.

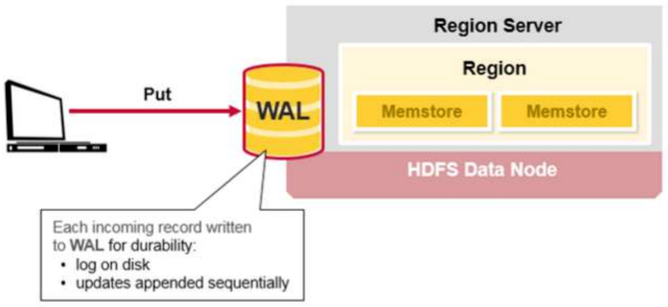


Region Server Components



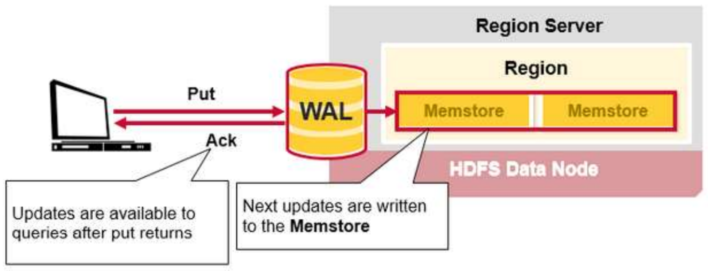
HBase Write Steps (1)

- On a client Put request, the data is first written to WAL:
  - Edits are appended to the end of the WAL file.
  - The WAL is used to recover not-yet-persisted data in case a server crashes.


Region Server
Region
Memstore
Memstore
HDFS Data Node
Put
WAL
Each incoming record written to WAL for durability:
• log on disk
• updates appended sequentially

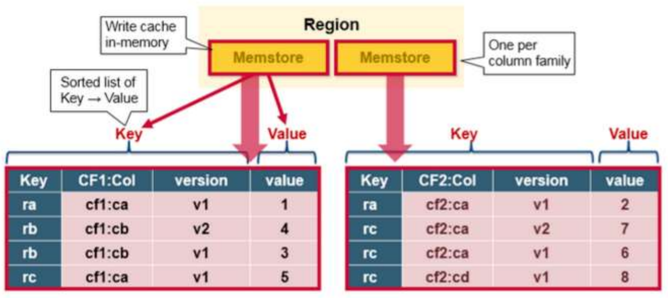
HBase Write Steps (2)

- Once the data is written to the WAL, it is placed in the MemStore.
- Then, the put request acknowledgement returns to the client.


Region Server
Region
Memstore
Memstore
HDFS Data Node
Put
Ack
WAL
Updates are available to queries after put returns
Next updates are written to the Memstore

HBase MemStore

- The MemStore stores updates in memory as sorted KeyValues (same as it would be stored in an HFile).
- There is one MemStore per column family. The updates are sorted per column family.

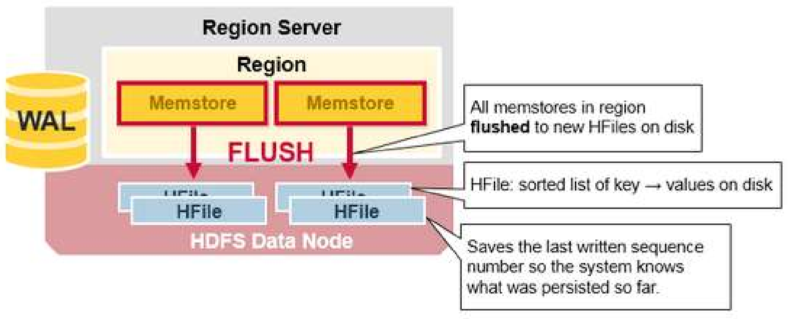


Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

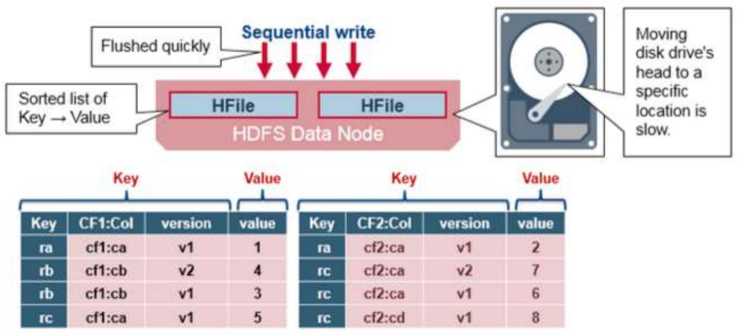
HBase Region Flush

- When the MemStore accumulates enough data, the entire set is flushed to a new HFile in HDFS.
- HBase uses multiple HFiles per column family, which contain the actual cells (KeyValue instances).


Region Server
Region
WAL
Memstore
Memstore
FLUSH
HFile
HFile
HDFS Data Node
All memstores in region **flushed** to new HFiles on disk
HFile: sorted list of key → values on disk
Saves the last written sequence number so the system knows what was persisted so far.

HBase HFile

- Data is flushed/stored in an HFile of sorted key/values.
- This is a sequential write. It is very fast, as it avoids moving the disk drive head.

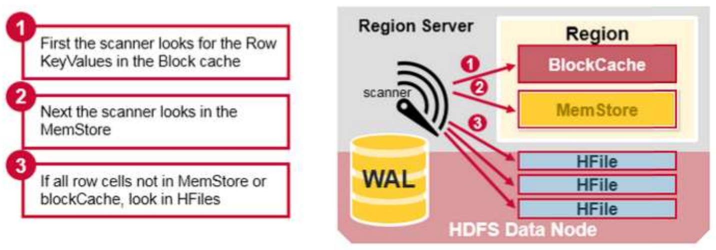

Flushed quickly
Sequential write
Sorted list of Key → Value
HFile
HFile
HDFS Data Node
Moving disk drive's head to a specific location is slow.
Key
Value
Key
Value

Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

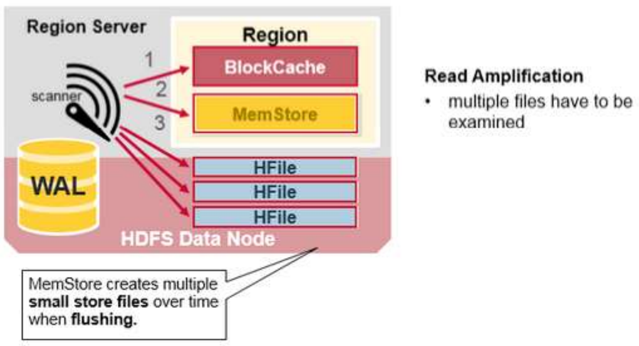
HBase Read Merge

- A Read merges Key Values from the block cache, MemStore, and HFiles in the following steps:
  1. First, the scanner looks for the Row cells in the Block Cache
  2. Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
  3. If still not found, load HFiles into memory.


1
First the scanner looks for the Row KeyValues in the Block cache
2
Next the scanner looks in the MemStore
3
If all row cells not in MemStore or blockCache, look in HFiles
Region Server
Region
BlockCache
MemStore
scanner
WAL
HFile
HFile
HFile
HDFS Data Node

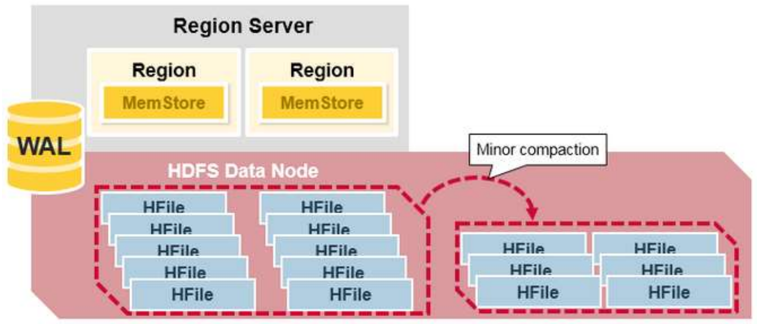
HBase Read Merge

- There may be many HFiles per MemStore. For a read, multiple files may have to be examined. This is called read amplification.


Region Server
Region
BlockCache
MemStore
scanner
1
2
3
WAL
HFile
HFile
HFile
HDFS Data Node
Read Amplification
• multiple files have to be examined
MemStore creates multiple small store files over time when flushing.

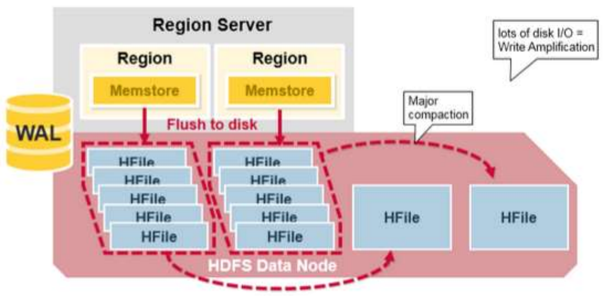
HBase Minor Compaction

- HBase will automatically pick some smaller HFiles and rewrite them into fewer bigger HFiles (minor compaction).
- Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.



HBase Major Compaction

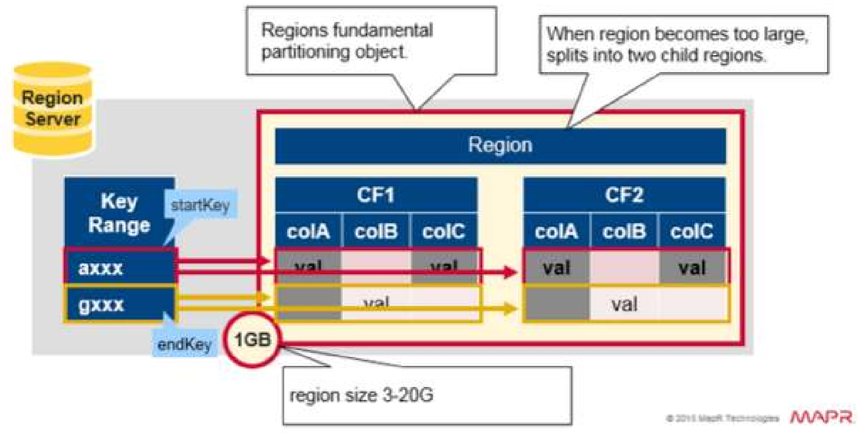
- Major compaction merges and rewrites all the HFiles in a region to one HFile per column family. This improves read performance. However, lots of disk I/O and network traffic might occur. This is called write amplification.



Regions Review

- A table can be divided horizontally into one or more regions.
- A region contains a contiguous, sorted range of rows between a start key and an end key.
- Each region is 1GB in size (default)
- A region is served to the client by a RegionServer
- A region server can serve about 1,000 regions (which may belong to the same table or different tables)

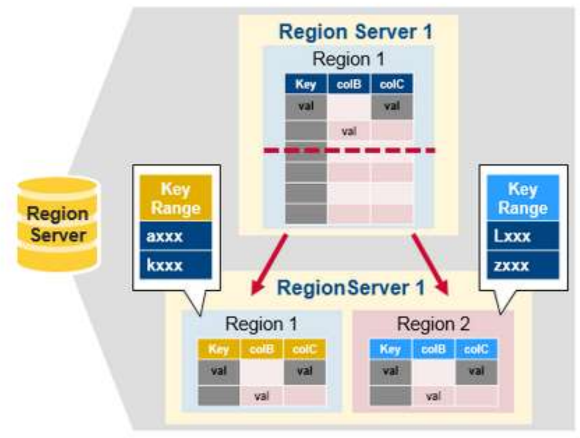
Regions Review


Regions fundamental partitioning object.
When region becomes too large, splits into two child regions.
Region Server
Region
CF1
CF2
colA
colB
colC
val
Key Range
startKey
axxx
gxxx
endKey
1GB
region size 3-20G
MAPR

Region Split

- Initially there is one region per table.
- When a region grows too large, it splits into two child regions.
- Both child regions are opened in parallel on the same Region server, and then the split is reported to the HMaster.
- For load balancing, the HMaster may schedule for new regions to be moved off to other servers.

Region Split

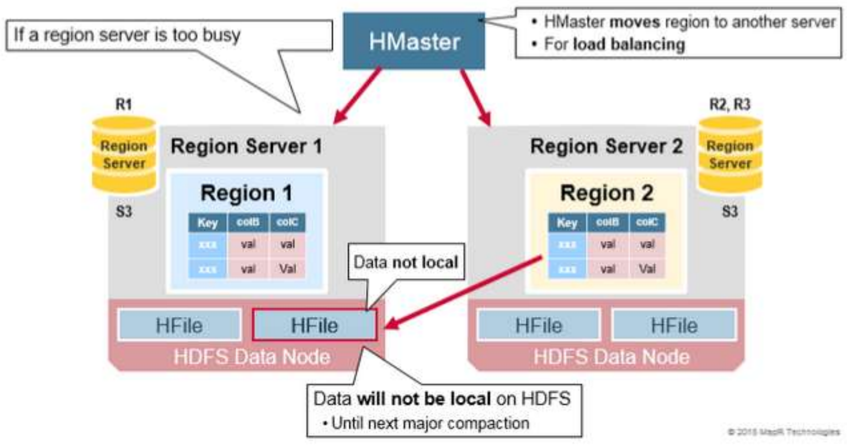


when region size > hbase.hregion.max. filesize → split

Read Load Balancing

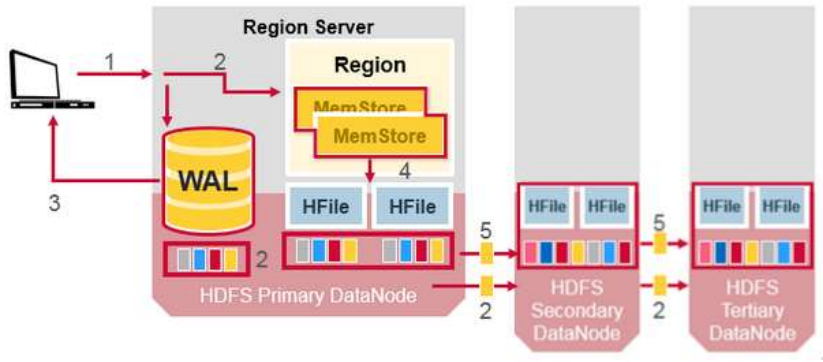
- Splitting happens initially on the same region server.
- HMaster may schedule for new regions to be moved off to other servers for load balancing.
- The new Region server serves data from a remote HDFS node until a major compaction moves the data files to the Regions server's local node.
- HBase data is local when it is written, but when a region is moved (for load balancing or recovery), it is not local until major compaction.

Read Load Balancing



HDFS Data Replication

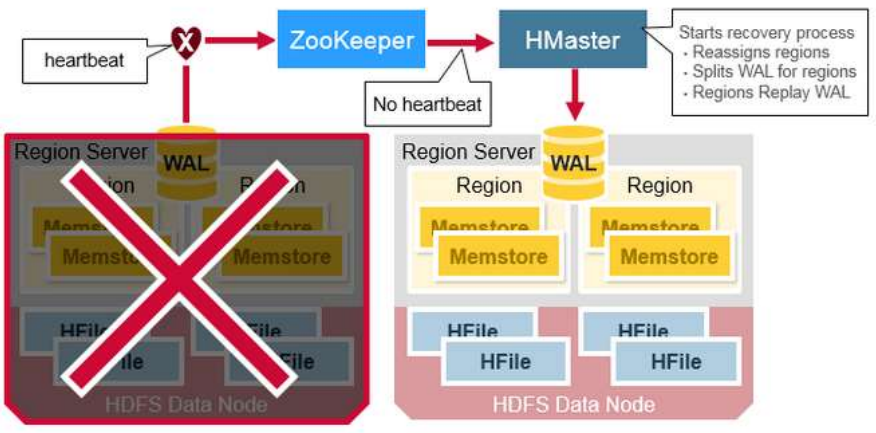
- HBase relies on HDFS to provide the data safety.
- HDFS replicates the WAL and HFile blocks automatically.



HBase Crash Recovery

- When a RegionServer fails, crashed Regions are unavailable.
- Zookeeper will detect node failure when it loses region server heart beats.
- HMaster will then be notified of Region Server failure.
- HMaster reassigns the regions from the crashed server to active Region servers.
- HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers' data nodes.
- Each Region Server then replays the WAL from the respective split WAL, to rebuild the Memstore for that region.

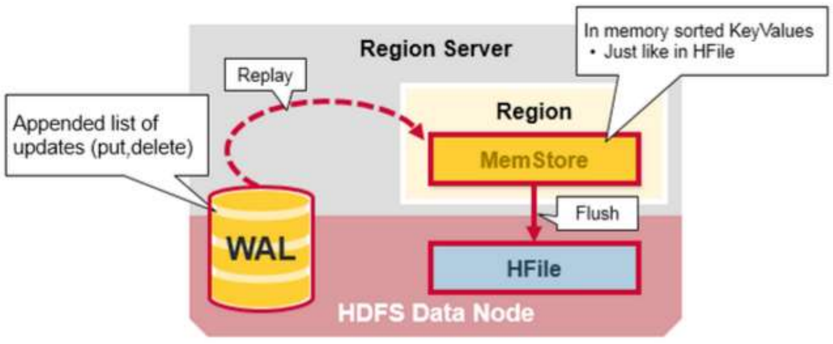
HBase Crash Recovery

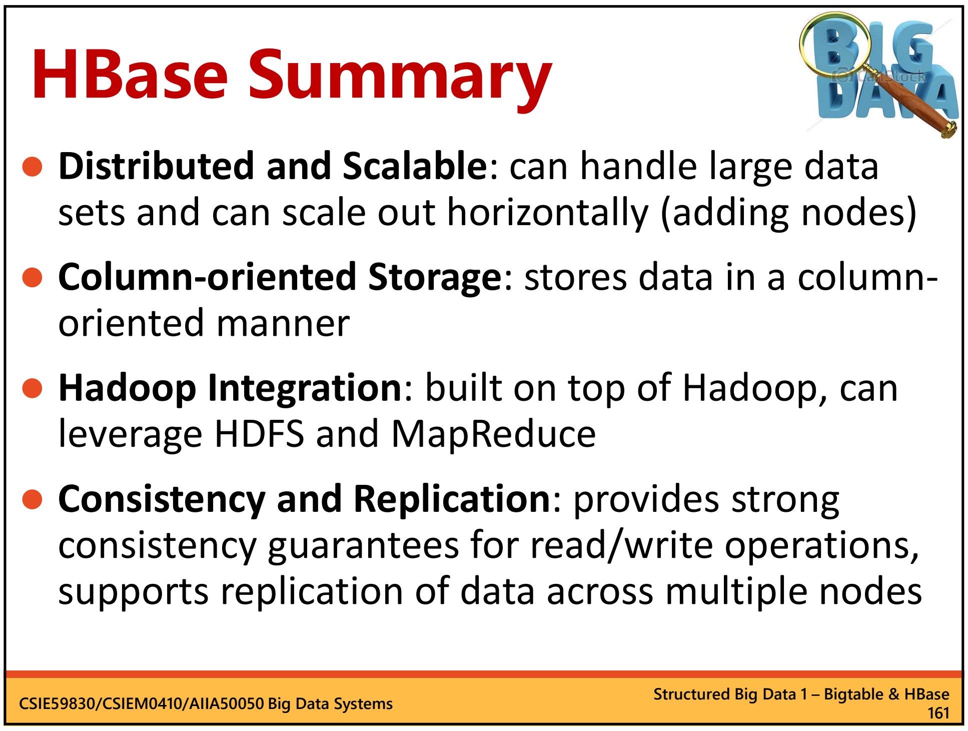


Data Recovery

- WAL files contain a list of edits, with one edit representing a single put or delete.
- Edits are written chronologically(按時間順序), additions are appended to the end of the WAL file that is stored on disk.
- On a failure when the data is still in memory and not persisted to an HFile, the WAL is replayed on the current MemStore.
- At the end, the MemStore is flush to write changes to an HFile.

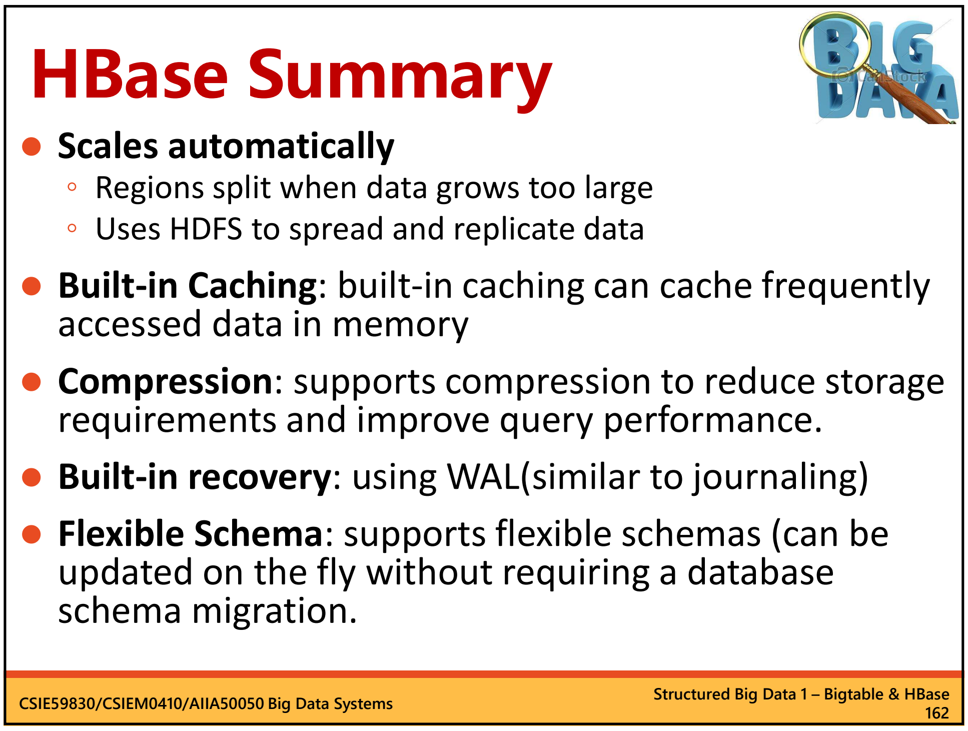
Data Recovery





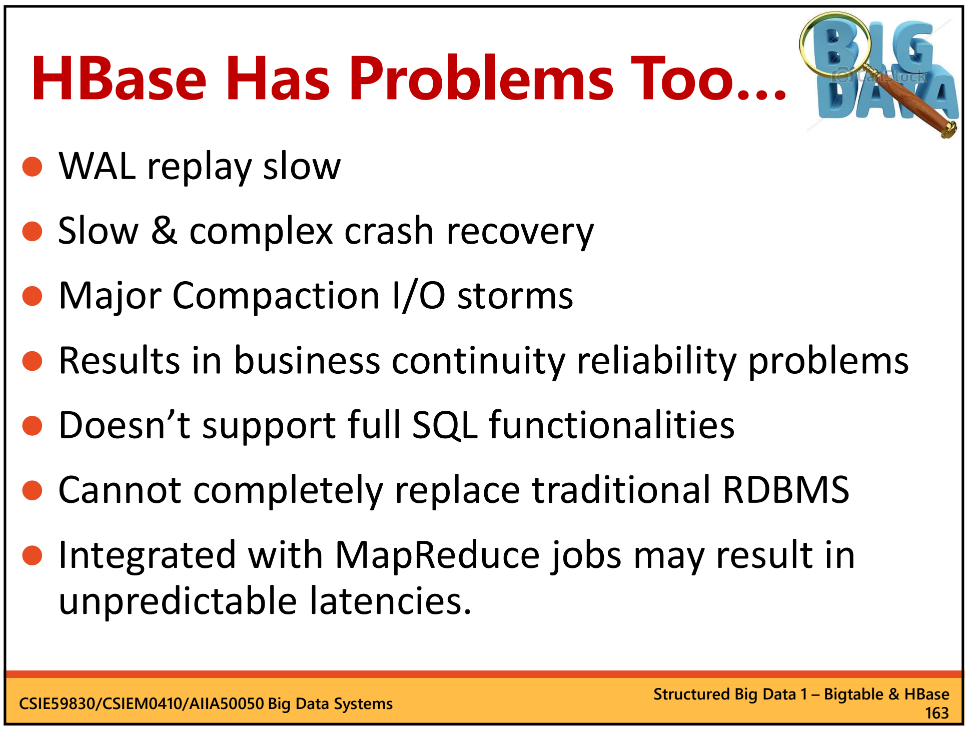
HBase Summary

- **Distributed and Scalable**: can handle large data sets and can scale out horizontally (adding nodes)
- **Column-oriented Storage**: stores data in a column-oriented manner
- **Hadoop Integration**: built on top of Hadoop, can leverage HDFS and MapReduce
- **Consistency and Replication**: provides strong consistency guarantees for read/write operations, supports replication of data across multiple nodes



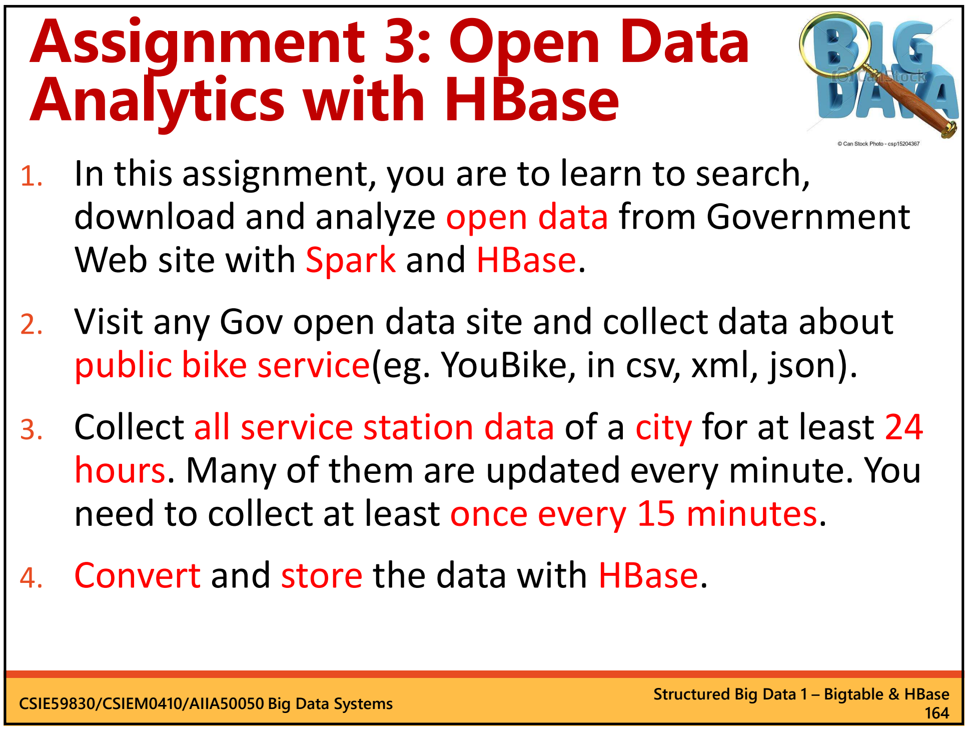
HBase Summary

- **Scales automatically**
  - Regions split when data grows too large
  - Uses HDFS to spread and replicate data
- **Built-in Caching**: built-in caching can cache frequently accessed data in memory
- **Compression**: supports compression to reduce storage requirements and improve query performance.
- **Built-in recovery**: using WAL(similar to journaling)
- **Flexible Schema**: supports flexible schemas (can be updated on the fly without requiring a database schema migration.



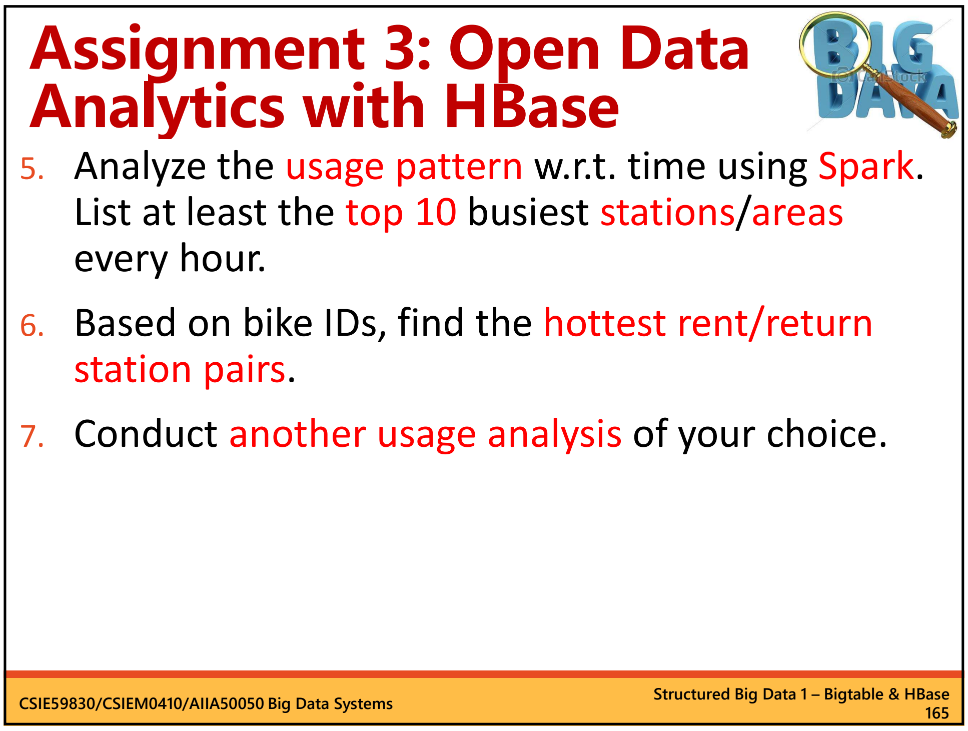
HBase Has Problems Too…

- WAL replay slow
- Slow & complex crash recovery
- Major Compaction I/O storms
- Results in business continuity reliability problems
- Doesn't support full SQL functionalities
- Cannot completely replace traditional RDBMS
- Integrated with MapReduce jobs may result in unpredictable latencies.



Assignment 3: Open Data Analytics with HBase

1. In this assignment, you are to learn to search, download and analyze open data from Government Web site with Spark and HBase.
2. Visit any Gov open data site and collect data about public bike service(eg. YouBike, in csv, xml, json).
3. Collect all service station data of a city for at least 24 hours. Many of them are updated every minute. You need to collect at least once every 15 minutes.
4. Convert and store the data with HBase.



Assignment 3: Open Data Analytics with HBase

5. Analyze the usage pattern w.r.t. time using Spark. List at least the top 10 busiest stations/areas every hour.
6. Based on bike IDs, find the hottest rent/return station pairs.
7. Conduct another usage analysis of your choice.