

Large Distributed Storage Systems

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

2023/4/13

Three parts in this unit

- ❑ Large distributed file systems
- ❑ **Large distributed storage systems**
- ❑ MapReduce programming model



Cloud Bigtable



Outline

- ❑ CAP Theorem
- ❑ Case Studies:
 - Google Bigtable
 - Apache HBase
 - Amazon Dynamo
 - Apache Cassandra
 - MongoDB

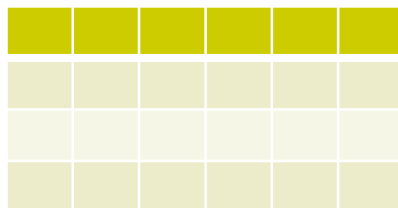
NoSQL (*non-SQL* → *Not Only SQL*)

- ❑ Many web-based applications have data in **unstructured format**, and with a huge volume of data, thus triggering the need of distributed NoSQL data stores.

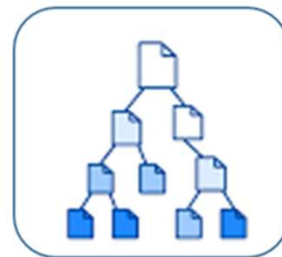
- ❑ Common categories:

- Key-value stores
 - information typically stored in JSON format.
- Column family stores
- Document stores
- Graph stores

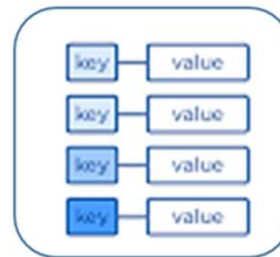
```
{  
  name: "yuh-jzer",  
  job: "teaching",  
  school: "ntu"  
}
```



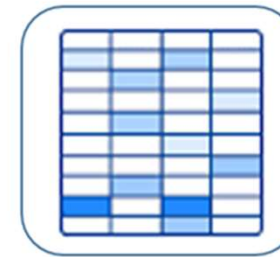
relational



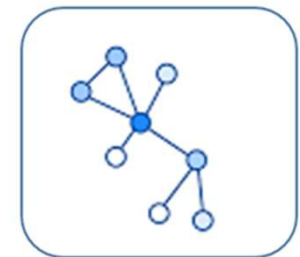
Document
Store



Key-Value
Store



Wide-Column
Store



Graph
Store

source: [Microsoft](#)

The **ACID** Property (for Transactions)

- ❑ **Atomicity:** a transaction is an atomic unit of processing and it is either performed entirely or not at all (“**all or nothing**”)
- ❑ **Consistency:** a transaction’s correct execution must take the database from one correct state to another
- ❑ **Isolation:** each transaction should appear to execute independently of other concurrent transactions.
- ❑ **Durability:** the effects of a completed transaction should always be persistent.

ACID property rules the RDBMS world where data objects are “monolithic” and replicas of them typically do not spread across wide area networks.

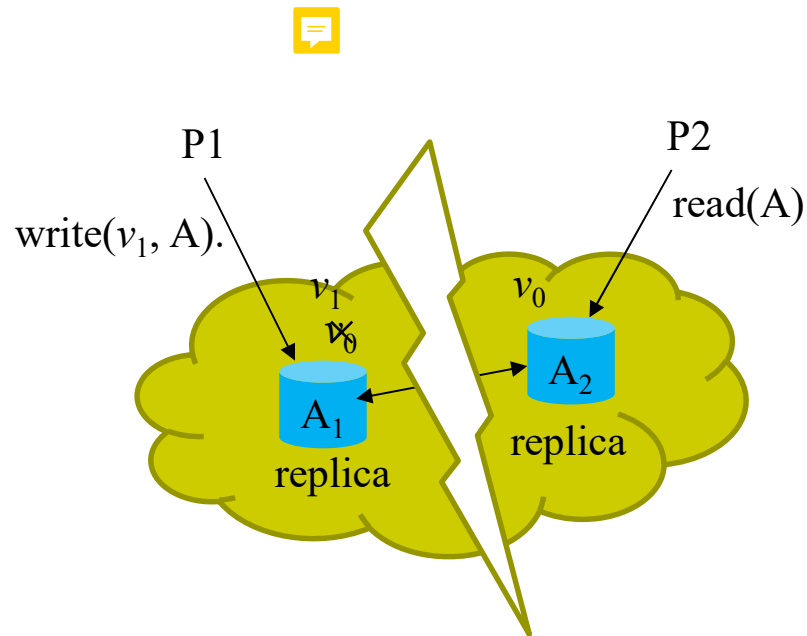
However, for Internet-scale services (e.g., shopping cart+order+payment service) in which network failures are not unusual, it is **impractical to adopt the “all-or-nothing”** concept of transactions to operations of data objects in the services.

CAP Theorem

- ❑ **Consistency**: every read receives the most recent writes or an error.
- ❑ **Availability**: every request received by a non-faulty node in the system must result in a response.
- ❑ **Partition tolerance**: the system continues to operate despite network partition.
- ❑ **CAP theorem**: it is impossible for a distributed system to simultaneously provide more than two out of the three guarantees.
 - The conjecture was made by Eric Brewer in his invited talk at PODC 2000, and was later formalized and confirmed by Seth Gilbert and Nancy Lynch in 2002.

Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, S. Gilbert & N. Lynch, ACM SIGACT News 2002.06.

Proof. of the CAP Theorem

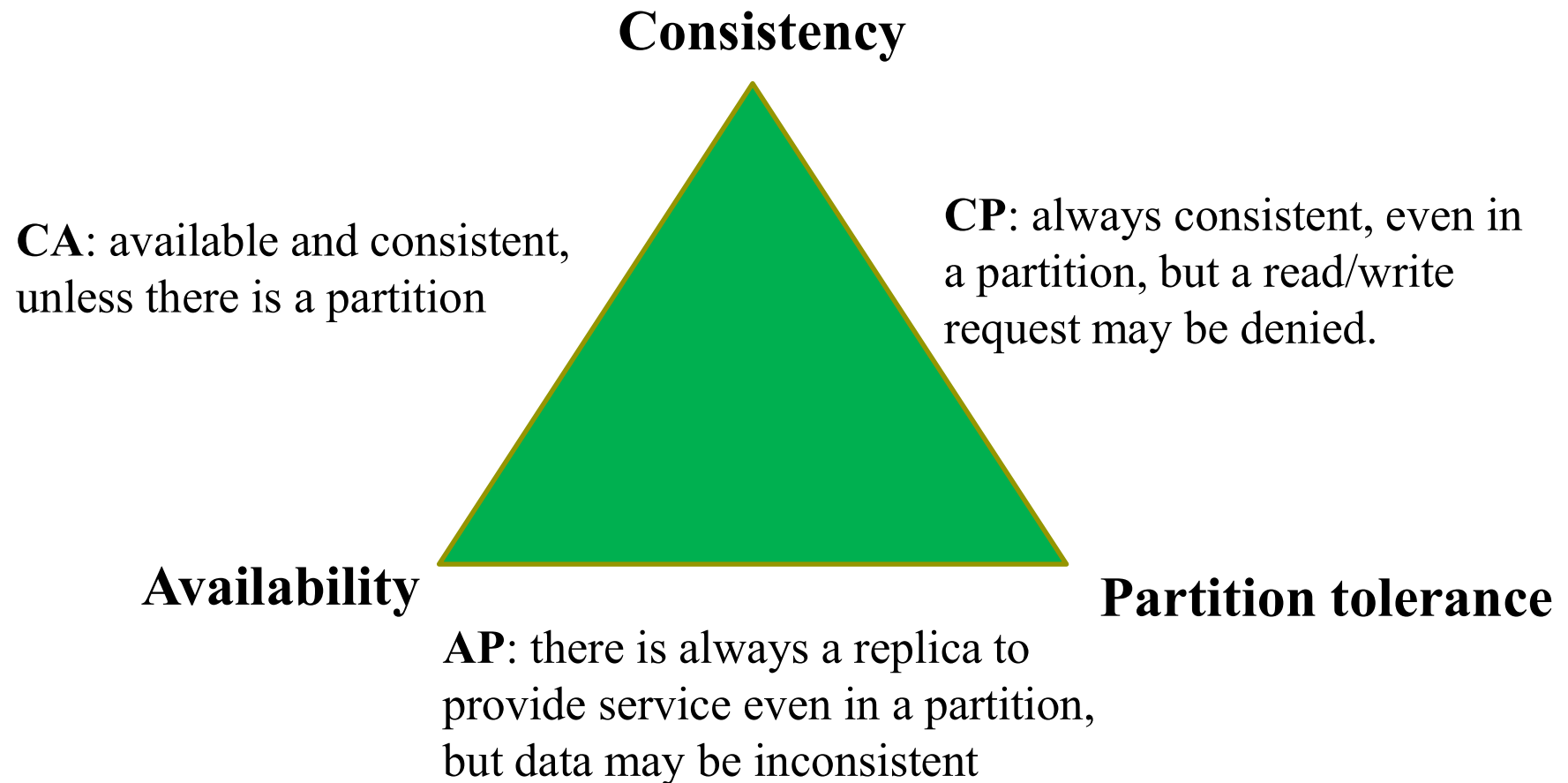


- Suppose each replica has value v_0 .
- Network partitioned. The system should continually to function.
- P1 invokes a write(v_1 , A). By the availability requirement, the operation must eventually terminates, and updates some replica with v_1
- P2 invokes a read(A). By the availability requirement, the operation must eventually terminates. Since no messages delivered between the two partitioned networks, A₂'s version remains to be v_0 , and so P2 obtains v_0 for its read.
- Then, this violates the consistency requirement.

Some trivial algorithms for guaranteeing any two requirements

- ❑ CP: Consistency + Partition tolerance
- ❑ CA: Consistency + Availability
- ❑ AP: Availability + Partition tolerance

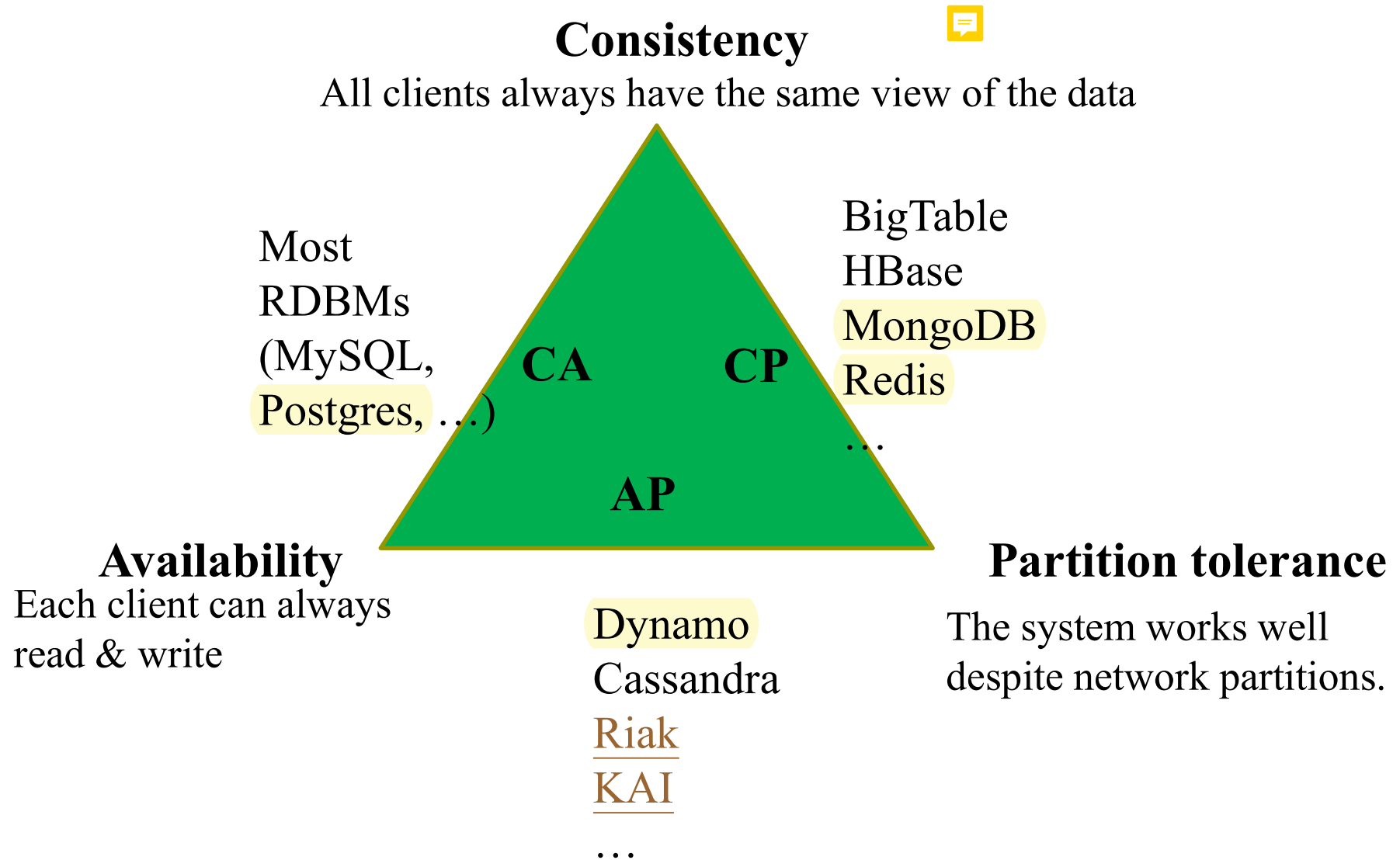
Need to sacrifice one of C, A, P



If a storage system needs to be fault-tolerant (w.r.t. network failures), it has to choose in between availability and consistency.

Usually, one would choose availability over consistency, why?

CAP in Database & Storage Systems



Doesn't mean **totally give up** one of them!

- ❑ Every system should be designed to ensure both C and A in normal situation
 - When a partition occurs the decision among C and A can be taken
 - When the partition is resolved the system must take actions to **resolve conflicts** that occur during the partition

Consistency/Latency Tradeoff

To achieve high availability,
data/services must be replicated

Consistency requires
communication, and the
stronger the consistency
imposed, the higher the
latency required.




High availability is a strong
requirement for large distributed
database/storage systems

Replication brings consistency
issues

CAP does not explicitly talk about latency...
... but latency is crucial to get the essence of CAP

BASE

❑ **B**asically **A**vailable, **S**oft State, **E**ventually consistent system

- **Basically Available**: reads/writes are available as much as possible but there is no guarantee on consistency
- **Soft State**: the state of the system might be stale at certain times
- **Eventually Consistent**: the system eventually converges to a consistent state 

The system must reconcile differences between replicas:

- **Anti-entropy**: exchanging versions or updates of data between servers

BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in availability.

■ **Reconciliation**: choosing an appropriate final state when concurrent updates have occurred

在化學裡，ACID是指“酸”，BASE是“鹼”

Google Bigtable and Chubby

Reading:

1. Bigtable: A Distributed Storage System for Structured Data, F. Chang, et al., ACMTOCS 2008.
2. The Chubby lock service for loosely-coupled distributed systems, M. Burrows, OSDI 2006.
3. Overview of Cloud Bigtable, Google

Bigtables

- ❑ Highly available distributed storage for structured data, e.g.,
 - URLs: content, metadata, links, anchors, page rank
 - Geography: roads, satellite images, points of interest, annotations
 - Used in Google for
 - Google Analytics, Google Finance, Personalized search, Blogger.com, Orkut, Google Code hosting, YouTube, Gmail, Google Earth & Google Maps, ...
- ❑ NoSQL
- ❑ Logically a (sparse) row-column table
- ❑ Physically stored by key-value (with **row×column as keys** for mapping to cells, or **row keys** for mapping to columns of cells)
 - Google used Sawzall (replaced by go (Lingo, Log in go), 2015) to process data in Bigtables. <https://golang.org>
- ❑ Distributed Multi-level map



Cloud Bigtable

Goal

❑ Scalable & Large scale

- Petabytes of data across thousands of servers
 - E.g., Billions of URLs with many versions per page
 - 100TB+ satellite image data
- Hundreds of millions of users
- Millions of read/write per second

❑ Self-managing

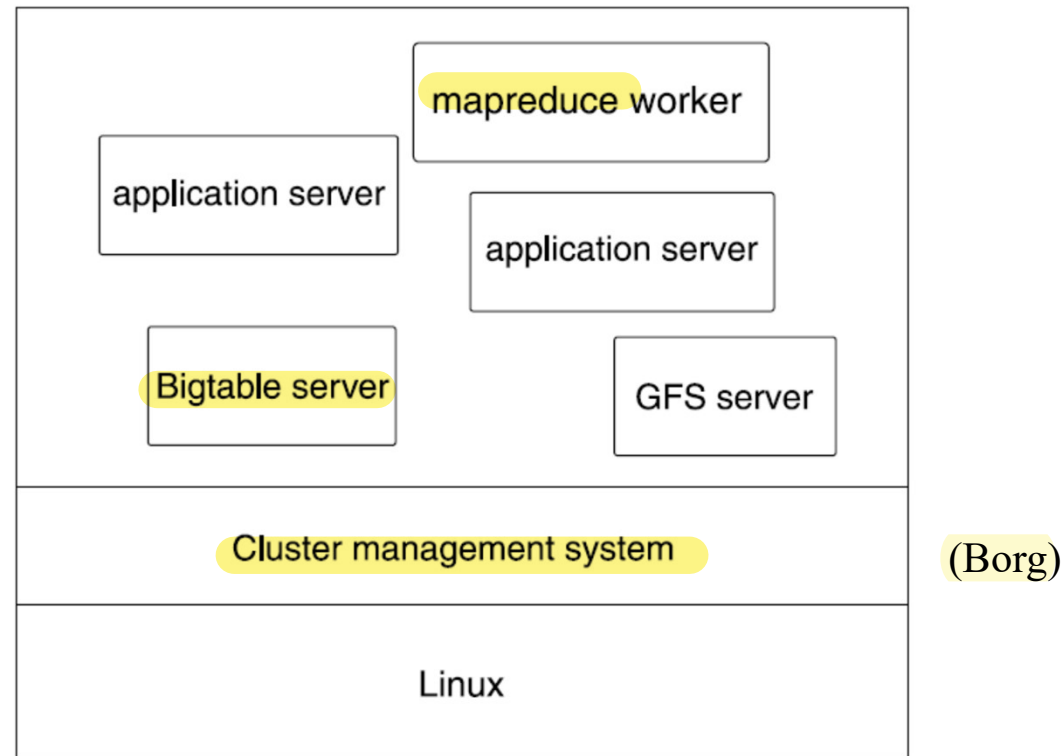
- Servers can be added/removed dynamically
- Servers adjust to load balancing

❑ Fault-tolerant, persistent

- provided by GFS & Chubby
- Each table can be configured for replication to multiple Bigtable clusters in different data centers

❑ Eventual consistency model

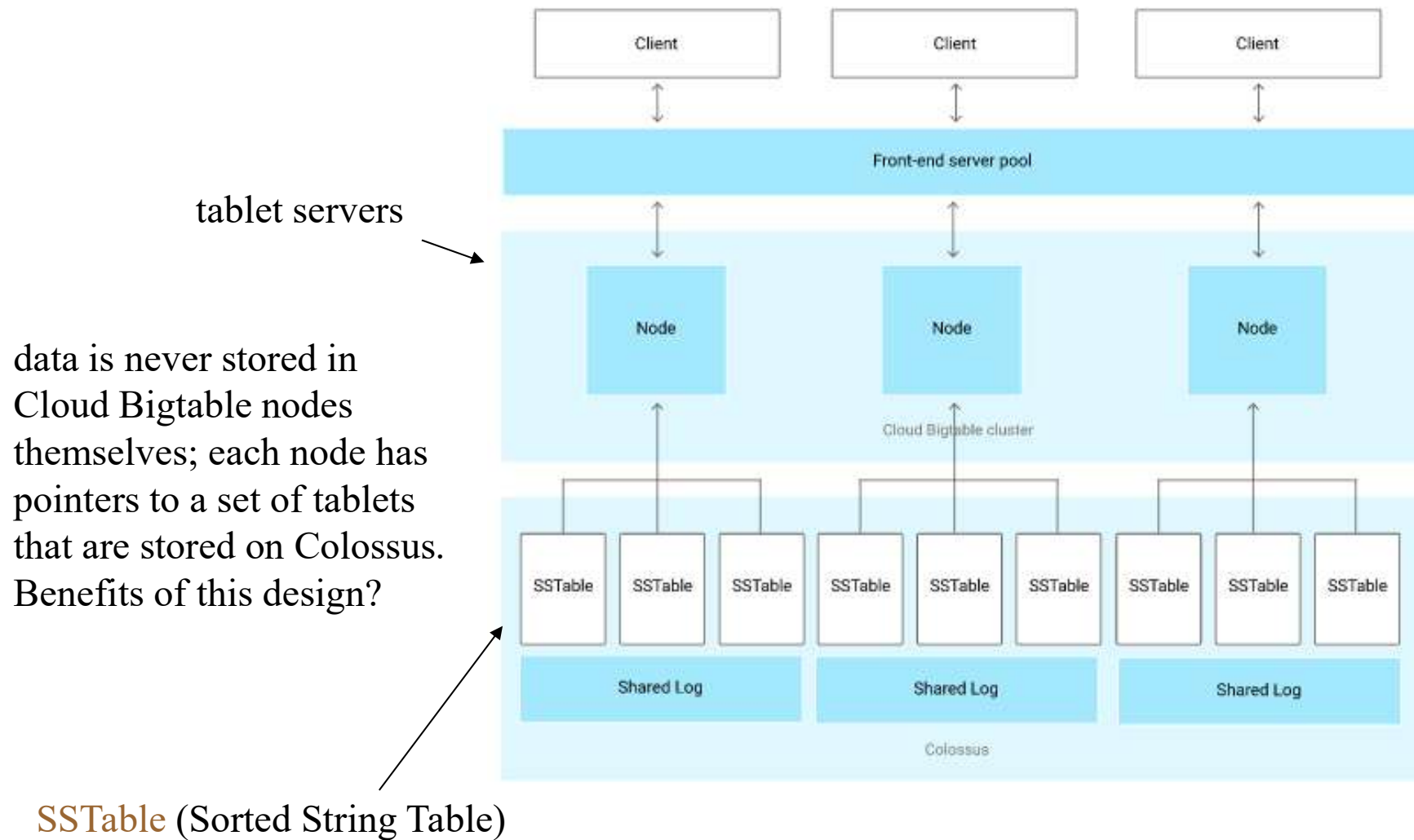
MapReduce, Bigtable & GFS



A typical set of processes that run on a Google machine. A machine typically runs many jobs from many different users.

source: [Bigtable: A Distributed Storage System for Structured Data](#), F. Chang, et al., ACMTOCS 2008.

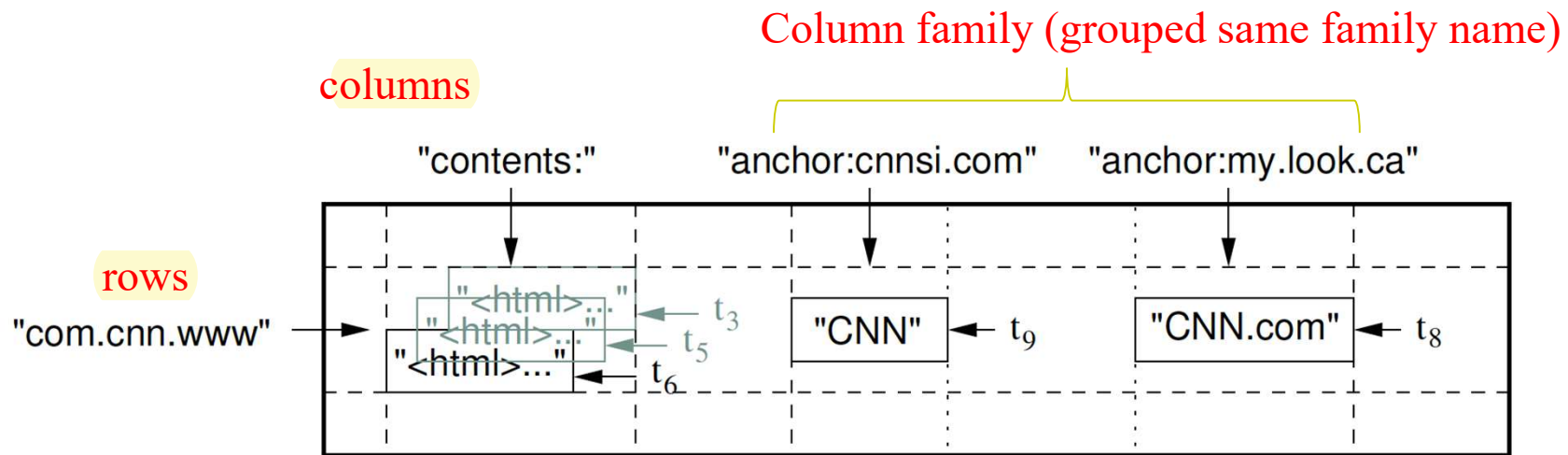
Cloud Bigtable Architecture



Source: [Overview of Cloud Bigtable](#), Google

Bigtable

- a sparse, distributed, persistent multidimensional table
 - (row:string, column:string, time:int64) → cell content



source: [Bigtable: A Distributed Storage System for Structured Data](#), F. Chang, et al., ACMTOCS 2008.

key-value logical view

Row key	Value
"com.cnn.www"	contents: {"<html>..."@t3, "<html>..."@t5, ...} anchor: {"cnnsi.com": "CNN", "my.look.ca": "CNN.com"@t8}
...	...

Column Families

- ❑ Column Family
 - Group of column keys
 - Basic unit of data access
 - Data in a column family is typically of the same type
 - Data within the same column family is compressed
- ❑ Identified by family:qualifier,

So, What is a **big table**?

Basically a key/value store in table format with three components

- **Row keys**
- Column families, including their garbage-collection policies
- Columns

where

- each table has only **one index**, the **row key**
- Rows are sorted lexicographically by row key, from the lowest to the highest byte string
- Column families are not stored in any specific order
- **Columns are grouped by column family** and sorted in lexicographic order within the column family
- The **intersection of a row and column** can contain multiple **timestamped cells**
- All operations are atomic at the row level
- Cloud Bigtable tables are sparse. A column doesn't take up any space in a row that doesn't use the column.

source: [Overview of Cloud Bigtable](#), Google

Some Schema Design Practices

❑ Tables

- Store datasets with similar schemas in the same table, rather than in separate tables.
 - Cloud Bigtable has a limit of 1,000 tables per instance

❑ Column families

- Put related columns in the same column family
- Create up to about 100 column families per table without experiencing performance degradation
- Choose short but meaningful names for your column families
- Put columns that have different data retention needs in different column families
 - Limit storage cost as garbage-collection policies are set at the column-family level, not at the column level

source: [Overview of Cloud Bigtable](#), Google

Some Schema Design Practices : Columns

- ❑ Treat column qualifiers as data



Jose	Fred:book-club	Gabriel:work	Hiroshi:tennis
Sofia	Hiroshi:work	Seo Yoon:school	Jakob:chess-club

vs.

Jose#1	Friend:Fred	Circle:book-club
Jose#2	Friend:Gabriel	Circle:work
Jose#3	Friend:Hiroshi	Circle:tennis
Sofia#1	Friend:Hiroshi	Circle:work
Sofia#2	Friend:Seo Yoon	Circle:school
Sofia#	Friend:Jakob	Circle:chess-club

source: [Overview of Cloud Bigtable](#), Google

Some Schema Design Practices: Columns

- ❑ Create as many columns as needed in the table.
 - (Cloud) Bigtable tables are sparse; there is no space penalty for a column that is not used in a row.
 - can have millions of columns in a table, as long as no row exceeds the maximum limit of 256 MB per row
- ❑ Avoid using too many columns in any single row
 - much more space-efficient to store that data in a single cell, rather than spreading the data across 1,024 cells that each contain 1 byte

source: [Overview of Cloud Bigtable](#), Google

Some Schema Design Practices: Rows

- ❑ Don't store more than 100 MB of data in a single row
- ❑ Keep all information for an entity in a single row
 - Atomicity is enforced in per row
- ❑ Store related entities in adjacent rows, to make reads more efficient

source: [Overview of Cloud Bigtable](#), Google

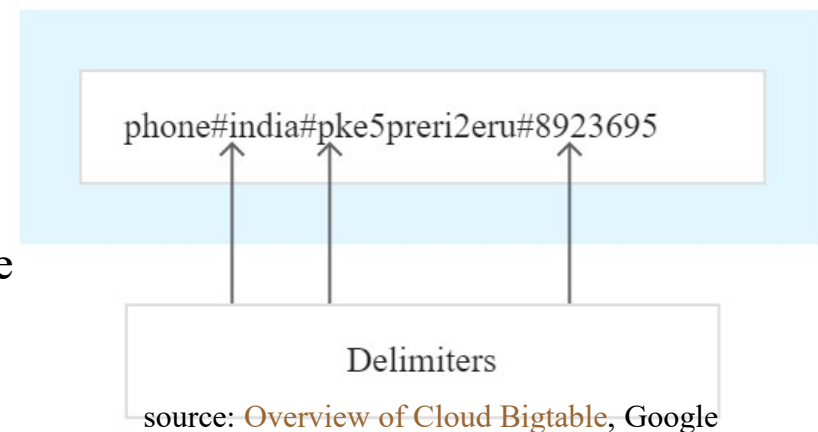
Some Schema Design Practices: Cells

- ❑ Don't store more than 10 MB of data in a single cell
 - a cell is the data stored for a given row and column with a unique timestamp
 - multiple cells (copies of data) can be stored at the intersection of that row and column
 - number of cells retained in a column is governed by the garbage collection policy that you set for the column's column family.

source: [Overview of Cloud Bigtable](#), Google

Schema Design Practices: Row keys

- ❑ Well-designed row keys get the best performance out of Cloud Bigtable
- ❑ The most efficient Cloud Bigtable queries retrieve data using one of the following:
 - Row key
 - Row key prefix
 - Range of rows defined by starting and ending row keys
- ❑ Keep your row keys short, 4 KB or less
- ❑ Store multiple delimited values in each row key
- ❑ Cloud Bigtable stores data **lexicographically**
 - $3 > 20$ but $20 > 03$.
 - Padding with leading zeros ensures the numbers are sorted numerically.



Schema Design Practices: Row keys

- ❑ Create a row key that makes it possible to retrieve a well-defined range of rows
- ❑ Use human-readable string values in row keys whenever possible
 - makes it easier to use the [Key Visualizer tool](#) to troubleshoot
- ❑ Design row keys that start with a common value and end with a granular value

tracks mobile device data:
device type#device ID#recorded day

phone#4c410523#20200501
phone#4c410523#20200502
tablet#a0b81f74#20200501
tablet#a0b81f74#20200502

asia#india#bangalore
asia#india#mumbai
asia#japan#okinawa
asia#japan#sapporo
southamerica#bolivia#cochabamba
southamerica#bolivia#lapaz
southamerica#chile#santiago
southamerica#chile#temuco

more [examples](#) for row keys:

Tables Are Typically Sparse

"follows" column family

	Follows			
Row Key	gashington	jadams	tjefferson	wmckinley
gashington		1		
jadams	1		1	
tjefferson	1	1		1
wmckinley			1	

Multiple versions


An example table for storing a social network for United States presidents:
each president can follow posts from other presidents.

source: [Overview of Cloud Bigtable](#), Google

Timestamps

- ❑ Each cell in a Bigtable can contain multiple versions of same data
 - Version indexed by a 64-bit timestamp: real time or assigned by client
- ❑ Per-column-family settings for garbage collection
 - Keep only latest n versions
 - Or keep only versions written since time
- ❑ Retrieve most recent version if no version specified

Tablets

- ❑ Table partitioned dynamically by rows into tablets
- ❑ Tablet = range of contiguous rows
 - Unit of distribution and load balancing
 - Nearby rows will usually be served by same server
 - Accessing nearby rows requires communication with small # of servers
 - Usually, 100-200 MB per tablet 
- ❑ Users can control related rows to be in same tablet by row keys
 - E.g., reverse URLs
 - “com.cnn.www”

Implementation: Components

1. Client side library

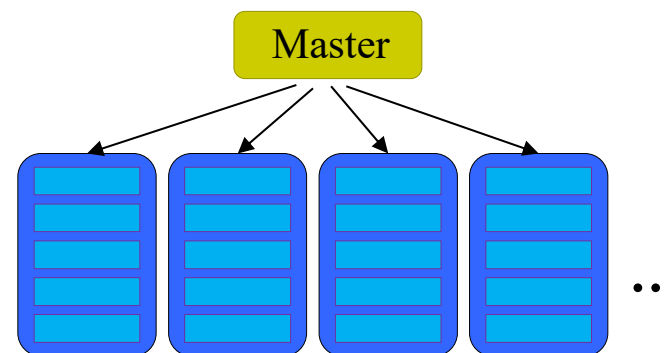


2. One master server

- Assigns tablets to tablet server
- Balances tablet server load
- Garbage collection of unneeded files in GFS
- Schema changes (table & column family creation)

3. Many tablet servers

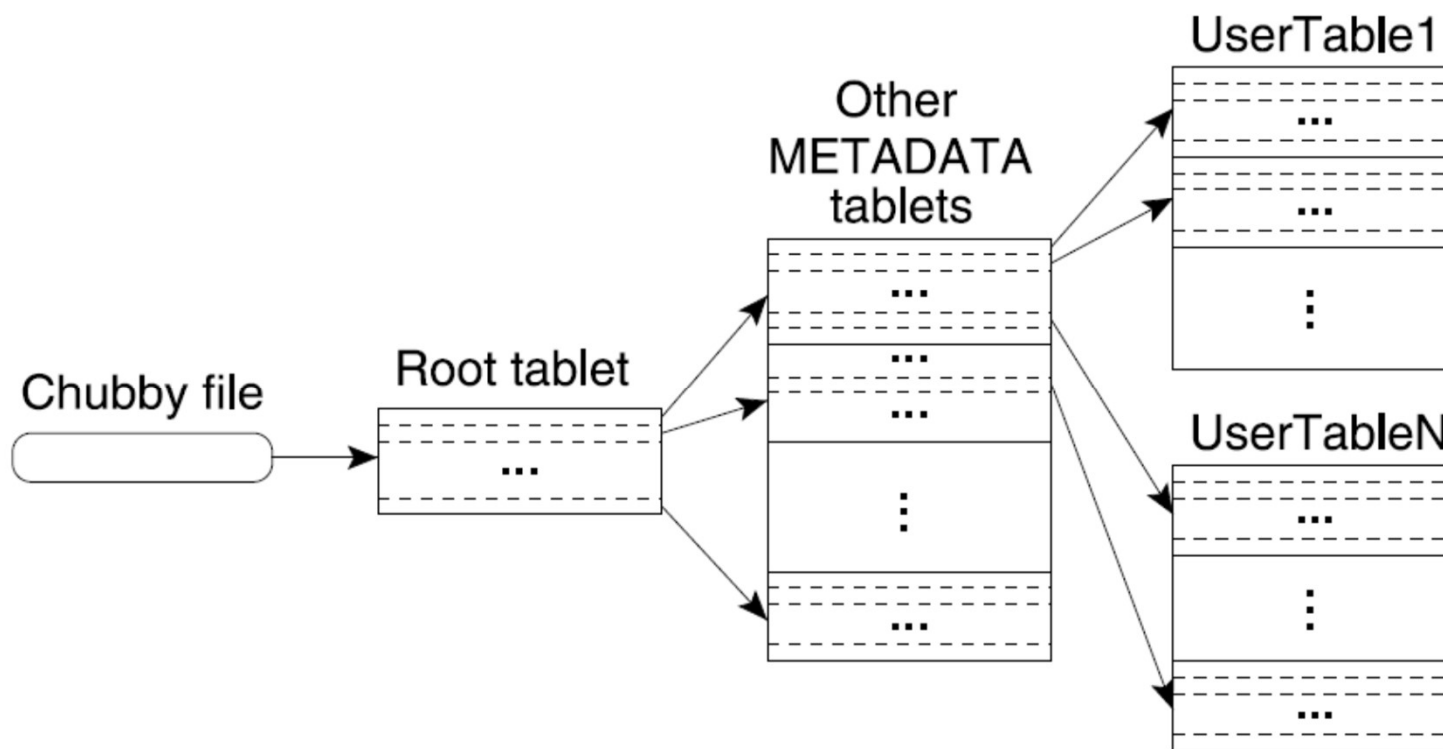
- coordinate requests to tablets
- can be added or removed dynamically
- each manages a set of tablets (typically 10-1,000 tablets/server)
- handles read/write requests to tablets
- splits tablets when too large



Implementation: Tablet location hierarchy

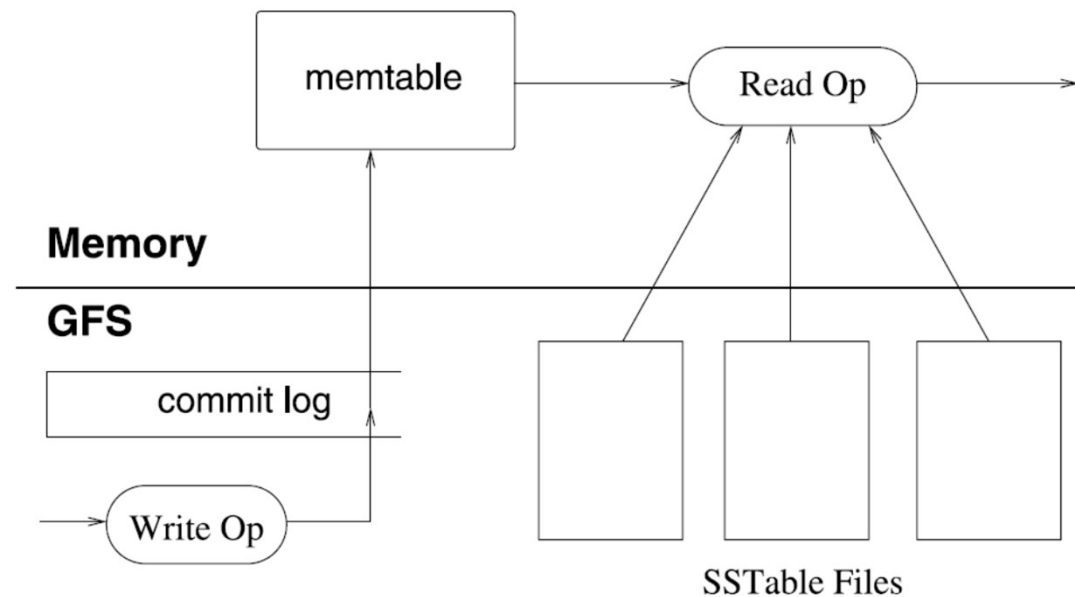
Three-level hierarchy

- Balanced structure similar to a B+ tree
- Root tablet contains location of all tablets in a special METADATA table
- Row key of METADATA table contains location of each tablet
 $f(\text{table_ID}, \text{end_row}) \Rightarrow \text{location of tablet}$



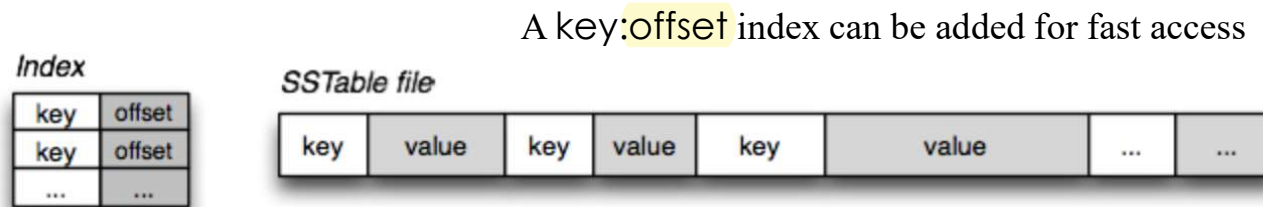
SSTables and Logs

- ❑ Updates are committed to a commit **log** that stores redo records
- ❑ Recently committed ones are stored in memory in a sorted buffer called a **memtable**
- ❑ Older updates are stored in a sequence of immutable **SSTables** (Sorted String Table) as chunks to GFS



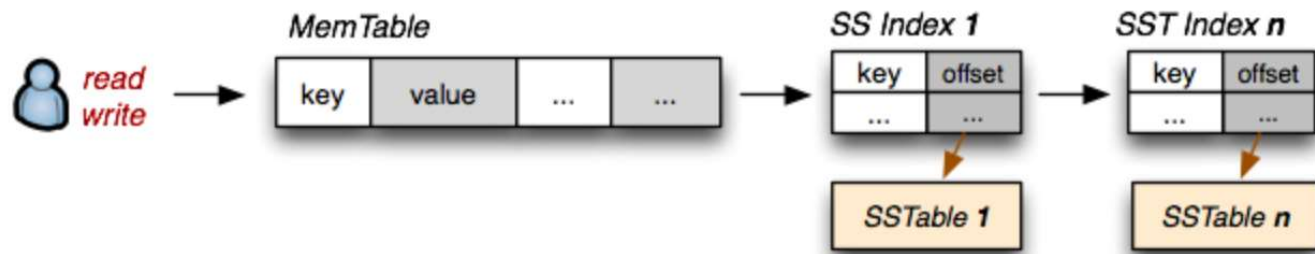
SSTable (Sorted String Table)

A simple abstraction to efficiently store large numbers of **key-value pairs** while optimizing for high throughput, sequential read/write workloads.



Once an SSTable is on disk it is effectively immutable because an insert or delete would require a large I/O rewrite of the file.

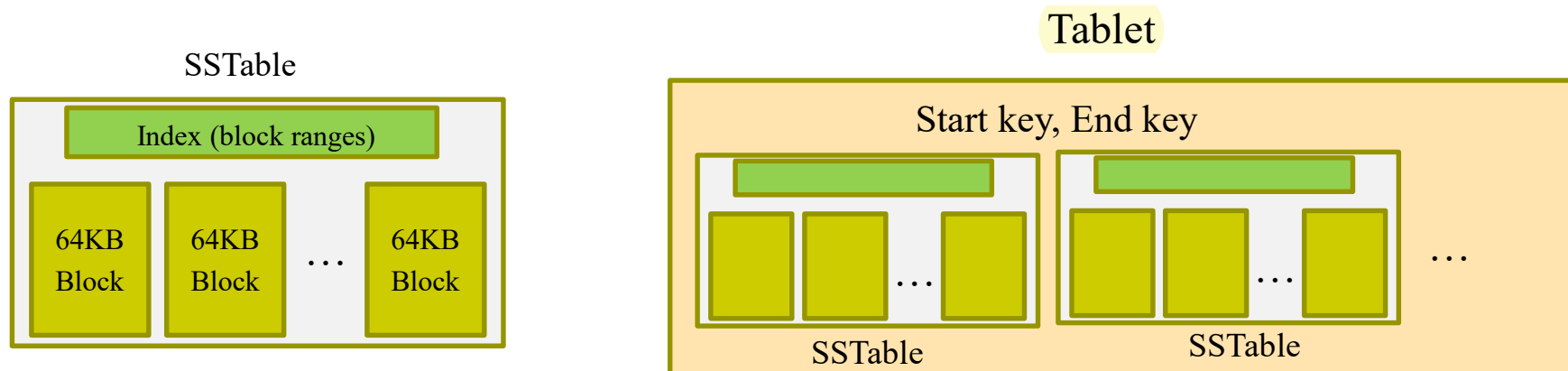
So perform random writes when the SSTable is in memory



source: [SSTable and Log Structured Storage: LevelDB, 2012](#)

SSTable

- ❑ SSTable provides a persistent, immutable, ordered map from keys to values
 - An SSTable contains a **sequence of blocks** (of default size 64KB), along with a **block index**
 - when SSTable is opened, index is loaded in memory for locating blocks from disk
- ❑ A tablet contains a number of SSTables with **adjacent rows**



Compaction

□ **Compaction:** write content to SSTables

- to keep memtable from growing too large after writes
- Reduce amount of data that has to be read from the commit log during recovery

■ **Minor compaction:**

- convert the memtable into a new SSTable
 - Too many SSTables might require read operations to merge updates from an arbitrary number of SSTables

■ **Merging compaction:**

- background read the contents of a few SSTables and the memtable, and writes out a new SSTable

■ **Major compaction:**

- rewrites SSTables into only one SSTable
- no deletion records, only live data

API Operations on Bigtable

- ❑ Create/delete tables & column families
- ❑ Change cluster, table, and column family metadata
 - (e.g., access control rights)
- ❑ Write or delete values in cells
- ❑ Read values from specific rows
- ❑ Iterate over a subset of data in a table
 - All members of a column family
 - Multiple column families
 - E.g., regular expressions, such as `anchor:*.cnm.com`
 - Multiple timestamps
 - Multiple rows
- ❑ Atomic read-modify-write row operations
- ❑ Allow clients to execute scripts (written in Sawzall) for processing data on the servers

[Ref. Cloud Bigtable APIs & reference](https://cloud.google.com/Bigtable/docs/apis)
<https://cloud.google.com/Bigtable/docs/apis>

Cloud Bigtable Tutorial

Introduction to Cloud Bigtable

<https://codelabs.developers.google.com/codelabs/cloud-bigtable-intro-java/index.html>

Datasets:

New York City Bus Data

Live data recorded from NYC Buses - Location, Time, Schedule & more

<https://www.kaggle.com/stoney71/new-york-city-transport-statistics>



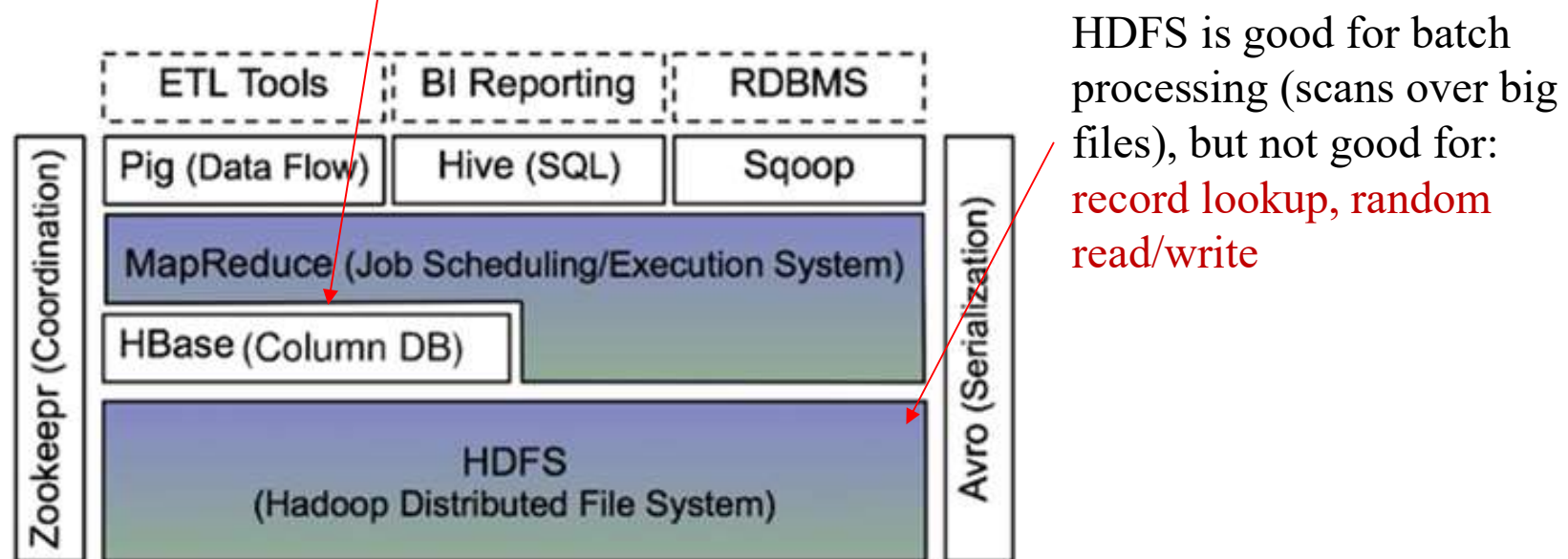
Apache HBase and ZooKeeper

Reading:

1. **HBase: the definitive guide: random access to your planet-size data**, Lars George, O'Reilly 2011 ([Google Book](#) provides some free content browsing)
2. **ZooKeeper: Wait-free Coordination for Internet-scale Systems**. P. Hunt, et al. USENIX 2010.

Apache HBase in Hadoop Ecosystem

- ❑ HBase: a distributed, fault-tolerant, highly scalable, no-SQL database, built on top of Hadoop Distributed File System (HDFS).
- ❑ Built on the Google Bigtable design with some small differences HBase files are internally stored in HDFS



ps. There are a number of views of Hadoop ecosystem. source: [hadoop-tutorial](http://hadoop-tutorial.com), Cloudera

HBase Data Organization

- ❑ Logically, data in HBase are organized in multidimensional **sparse tables** of **rows** and **columns**
 - each row has an arbitrary number of columns, which are grouped into column families
 - table schema only defines its column families
 - a family can consist of any number of columns
 - new **column-families cannot be added for an existing table**, but **columns within any pre-existing column-family can be added on-the-fly**
 - rows are sorted by **row key** (uninterrupted byte array)
 - a cell's content (uninterrupted byte array) is uniquely identified by
 - Table + Row-Key + **Column-Family**:Column +Timestamp (version)
 - row updates are atomic

Ref. [Hadoop-HBase for large-scale data](#), M. N. Vora 2011.



Tables and Regions

region 1	column family			column family	
	row key	personal_data		demographic	...
	PersonalID	Name	Address	BirthDate	Gender
	A000110121	Y. Huang	KeeLung Road, ...	1980.01.01	M

region 2	row key	personal_data		demographic	...
	PersonalID	Name	Address	BirthDate	Gender
	N002101781	M. Chen

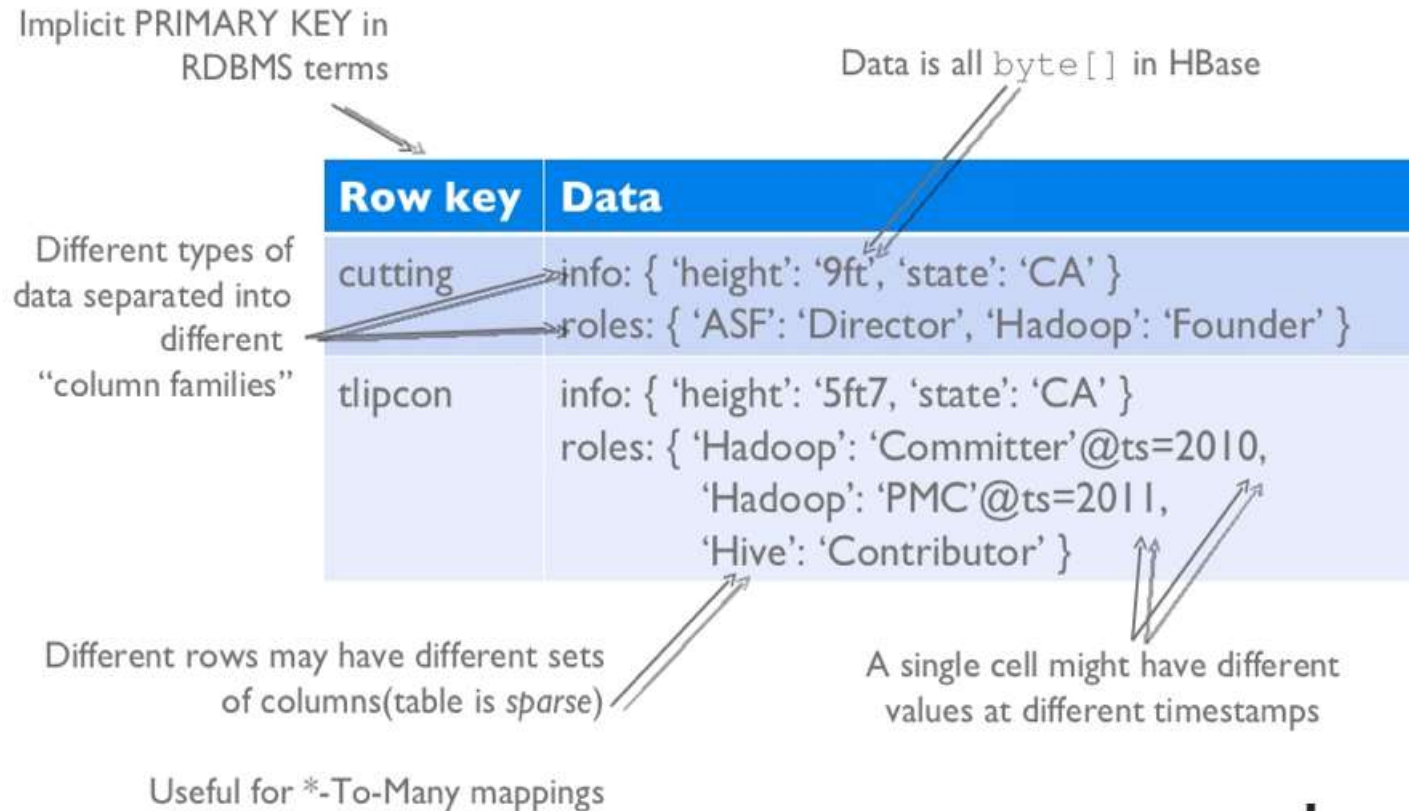
HBase has Dynamic Columns, as column names are encoded inside the cells

Different cells can have different columns

HBase tables are automatically partitioned horizontally into **regions**, each region comprising a subset of a table's rows

A region is defined by its first and last row, plus a randomly generated region identifier.

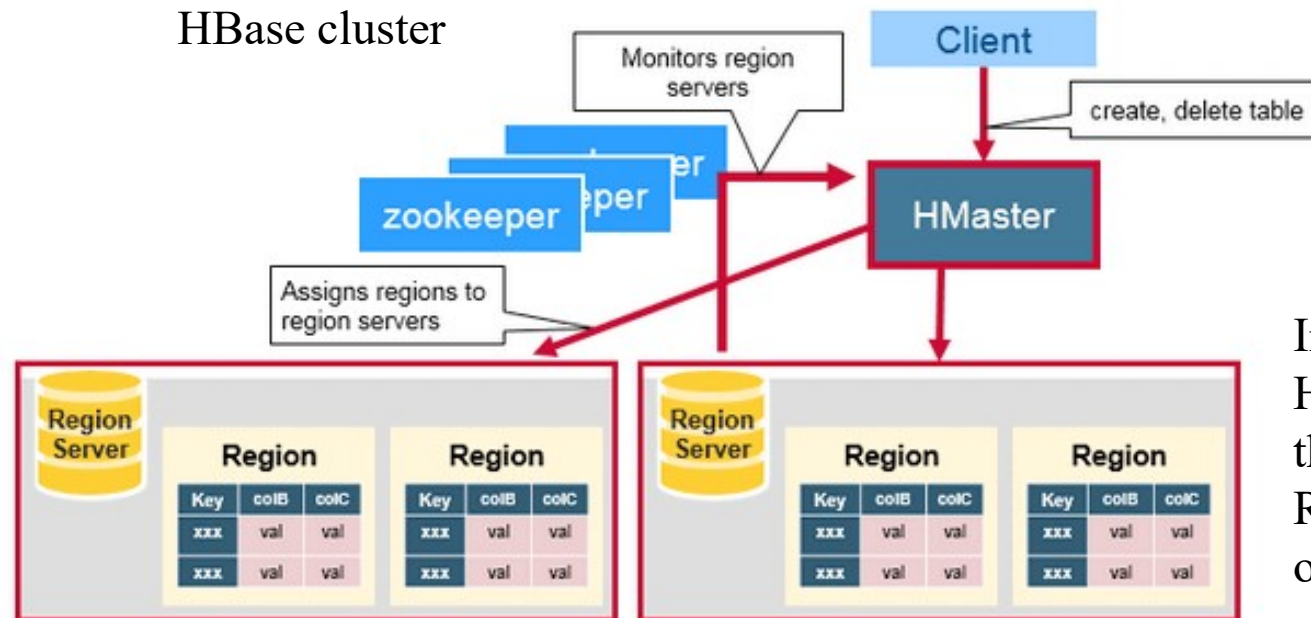
Logical View (Key-Value) of Tables



source: HBase, M. Eltabakh 2013, slideplayer.com/slide/1544896/

HBase Architecture

- ❑ Similar to HDFS and Map/Reduce, HBase also adopts master/slave architecture:
 - **HMaster** (master) is responsible for assigning regions to HRegionServers (slaves) and for recovering from HRegionServer failures.
 - **HRegionServer** is responsible for managing client requests for reading and writing.
 - uses Zookeeper (cf. Google's Chubby, to be discussed later) for management of HBase cluster



In a Hadoop cluster, HMaster typically runs on the NameNode, while RegionServer typically runs on the DataNode.

source: corejavaguru.com

HBase RegionServer

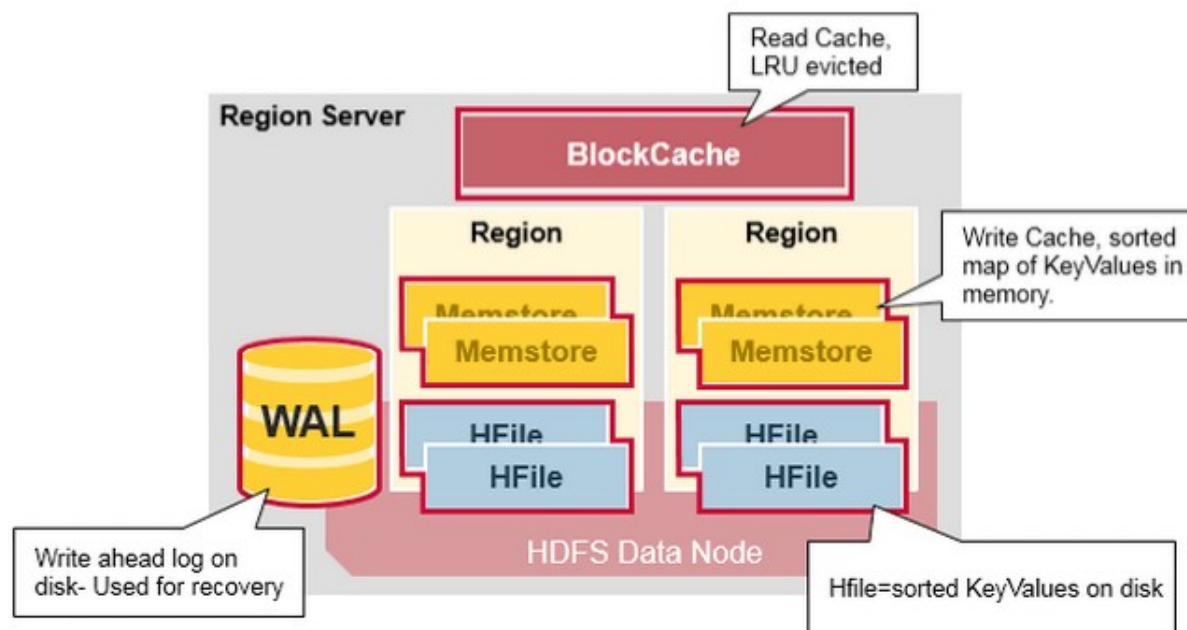
Row's data can be separated in multiple column families (CFs). Data of particular CF is stored in HStore which consists of Memstore and a set of HFiles.

Write-Ahead Log (Hlog): one per RegionServer, store write data that hasn't yet been persisted to permanent storage; used for failure recovery

BlockCache: read cache

MemStore: write cache for new data not yet been written to disk. One MemStore per column family per region.

Hfiles: files that store the rows as sorted KeyValues on disk (in HDFS)



source: corejavaguru.com

HBase Regions

In HBase, data is stored per column family: each column family has a read cache BlockCache and a write cache MemStore.

Data is read in blocks from the HDFS and cached in BlockCache.

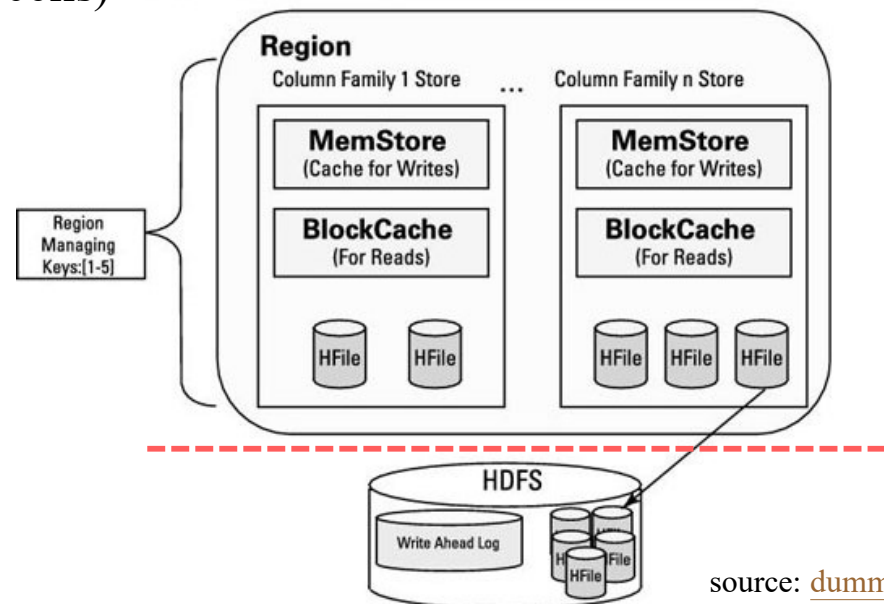
To ensure HBase writes are reliable, write/update data is first persisted to the WAL, which is stored in the HDFS, and then written to MemStore

At configurable intervals, key-value pairs stored in the MemStore are written to HFiles in the HDFS and then WAL entries are erased.

Minor compactions: combine recently written Hfiles into a larger one

Major Compactions: merge all the HFiles for a Region (per column family) and writes to a single Hfile (and remove the deleted cells)

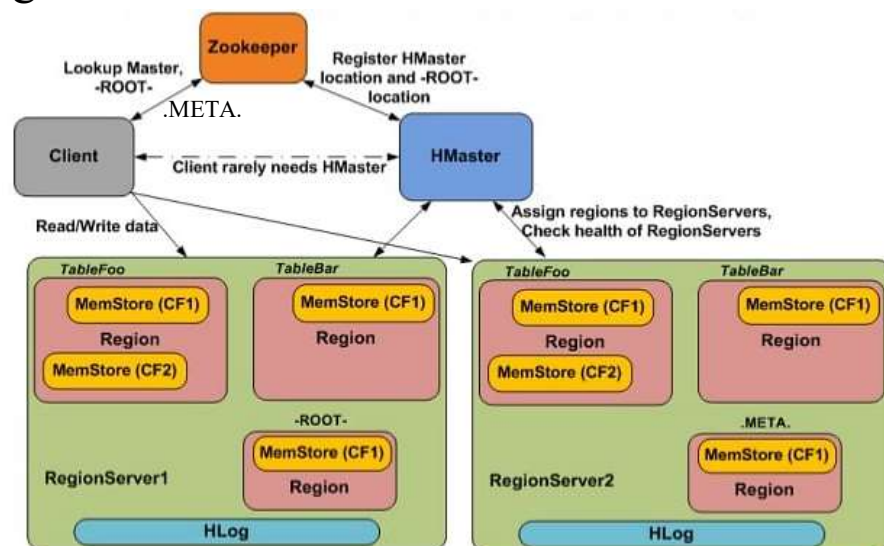
An HFile contains a multi-layered index like a b+tree, allowing HBase to seek to the data without having to read the whole file



source: dummies.com

ZooKeeper in HBase

- ❑ HBase uses ZooKeeper as a distributed **coordination service** for
 - Monitor servers
 - HMaster and Region servers register ZooKeeper and get notification of server failures or network partitions
 - clients use ZooKeeper to locate HMaster and Region servers
 - Maintain Configuration Information
 - -ROOT-: list of .META. tables (removed for Version 0.96.0 onwards)
 - .META. : mapping of Regions to Region Server, like a b tree with key-values
 - Key: region start key, region id
 - Values: RegionServer



source: [cloudera](https://cloudera.com/)

cloudera

Apache ZooKeeper

- ❑ A distributed, open-source **coordination service** for distributed applications
 - provide a simple and high performance kernel (APIs) for building more complex coordination primitives at the client, e.g.,
 - Application configuration, group membership, leader election, locking, ordering of operations, ...
 - Function likes Google Chubby, but abandoning the lock approaches so that operations can be **wait-free**
 - relaxed consistency guarantees

- ❑ cf. Chubby: provide coarse-grained locking as well as reliable storage for a loosely-coupled distributed system

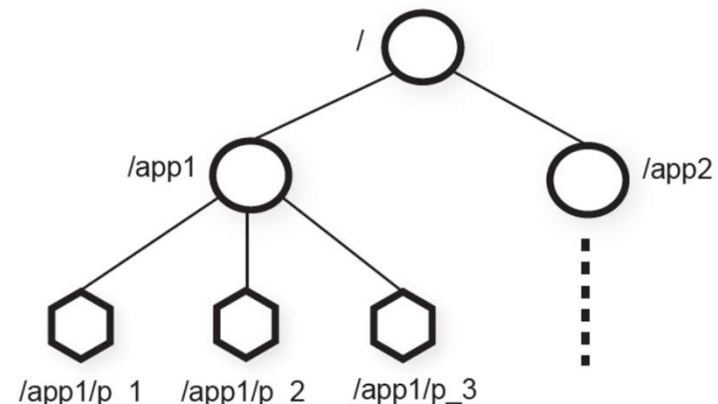
Ref. ZooKeeper: Wait-free Coordination for Internet-scale Systems. P. Hunt, et al. USENIX 2010.



Data model and the hierarchical namespace

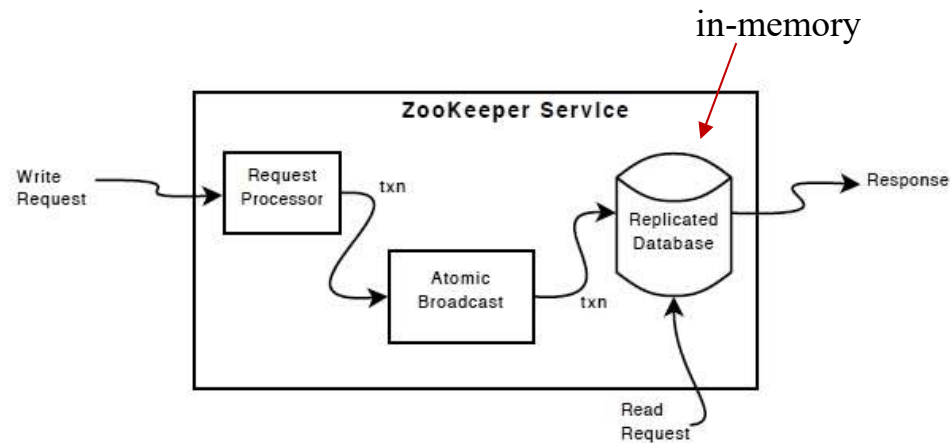
- ❑ Clients coordinate with each other by manipulating znodes through APIs.
- ❑ Znodes are organized hierarchically similarly to a standard file system.
 - Each node is identified by a path.
 - Types of znodes:
 - Regular: clients must delete them explicitly;
 - Ephemeral: ZooKeeper automatically removes them when their sessions terminate
 - A (TCP) connection to server from client is a session
 - Flags of Znode
 - Sequential flag
 - creating node receives a monotonically increasing counter appended to its name.
 - Watch Mechanism
 - Get notification
 - One time triggers

Data is kept in-memory to achieve high throughput and low latency

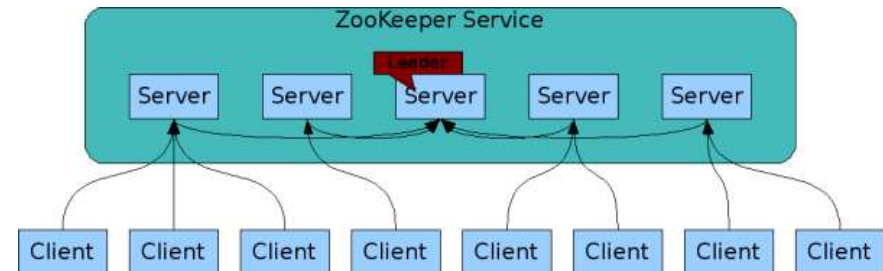


ZooKeeper hierarchical name space

ZooKeeper Architecture & Implementation



- Read requests are serviced from the local replica of each server database.
- Write requests are forwarded to the **leader** to process an agreement protocol among the servers.
- each update is timestamped reflecting the order of all ZooKeeper transactions. Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives.



- Failure model: fail crash, faulty process may recover
- One of the server is elected as the **leader**, the rest are **followers**
- Clients connect to exactly one server to submit its requests.
- Zookeeper service is available as long as a majority of the servers are available

Atomic Broadcast

- ❑ Atomic broadcast (total order broadcast): a broadcast where all correct processes in a system receive the same set of messages in the same order.
- ❑ Additional requirements
 - Validity: if a correct participant broadcasts a message, then all correct participants will eventually receive it.
 - Uniform Agreement: if one correct participant receives a message, then all correct participants will eventually receive that message.
 - Uniform Integrity: a message is received by each participant at most once, and only if it was previously broadcast.

Equivalent to the **Consensus** problem (to be discussed later in the course)

How to implement?

Zookeeper APIs

- ❑ `create(path, data, flags)`: creates a znode with path, data; flags enables a client to select the type of znode: regular, ephemeral, and set the sequential flag
- ❑ `delete(path, version)`: deletes the znode path if that znode is at the expected version;
- ❑ `exists(path, watch)`: test if znode path exists. The watch flag enables a client to set a watch on the znode;
- ❑ `getData(path, watch)`: returns the data and meta-data of the znode.
- ❑ `setData(path, data, version)`: writes data to znode path if the version number is the current version of the znode;
- ❑ `getChildren(path, watch)`: returns the children of a znode;
- ❑ `sync(path)`: waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. The path is currently ignored.

All methods have both a synchronous and an asynchronous version

ZooKeeper uses a **watch** mechanism to allow clients to receive timely notifications of changes without requiring polling.

Watches are one-time triggers associated with a session; they are unregistered once triggered or the session closes.

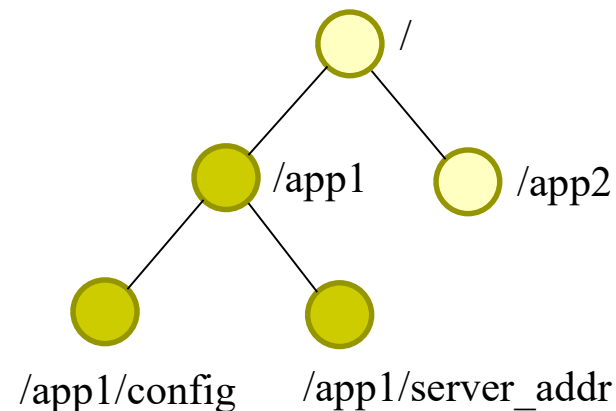
Some Primitives Implemented by the APIs

□ Configuration Management:

- By storing a configuration in a znode, z_c , starting processes can obtain their configuration by reading z_c with the watch flag set to true to receive update notification

□ Rendezvous

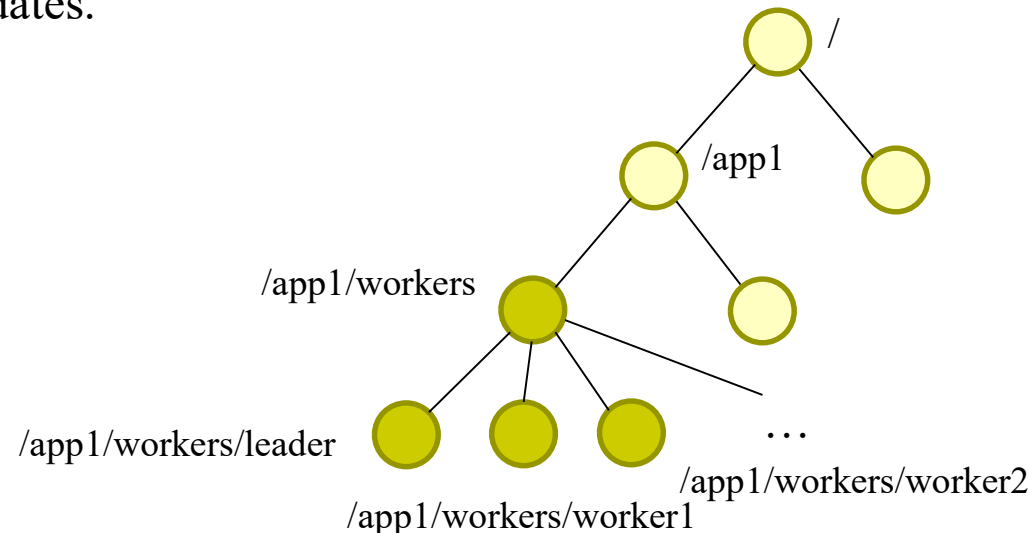
- Application: let a set of worker processes created by a scheduler know the master's communication address which is not known a priori.
- Solution: let the application client create a rendezvous znode, z_r , and pass the full pathname of z_r as a startup parameter of the master and worker processes. When the master starts, it fills in z_r with information about addresses and ports it is using. When workers start, they read z_r with watch set to true.



Some Primitives: Group Membership

□ Group Membership

- Designating a znode, z_g to represent the group.
- When a member process starts, it creates an **ephemeral** child znode under z_g .
 - If it has a unique name, the name is used as the name of the child znode;
 - otherwise, it can use the SEQUENTIAL flag to obtain a unique name assignment (appended with a monotonically increasing counter).
 - Other information (e.g., communication port) can be put into the node as well.
- By the ephemeral node property, if a process fails or ends, its representing znode under z_g is automatically removed. This then allows processes to obtain/monitor group information by simply listing the children of z_g , and use watch flag to obtain membership updates.



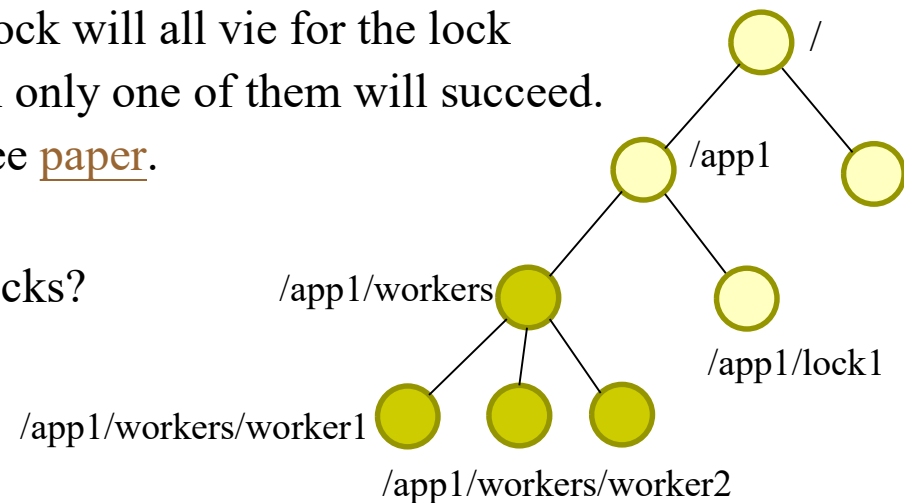
Some Primitives: Locks

□ Simple Locks

- Use a znode to represent the lock.
- To acquire a lock, a client tries to create the designated znode with the EPHEMERAL flag. A success means acquiring the lock. Otherwise, the client can read the znode with the watch flag set to be notified.
- A client releases the lock when it dies or explicitly deletes the znode. Other clients that are waiting for a lock try again to acquire a lock once they observe the znode being deleted.

□ Two problems with above implementation:

- Herd Effect
 - Processes waiting to acquire a lock will all vie for the lock when it is released, even though only one of them will succeed.
 - Remedy: order their requests; see [paper](#).
- Only exclusive locks
 - How to implement read/write locks?
see [paper](#).



Some More Primitives

❑ Double barriers

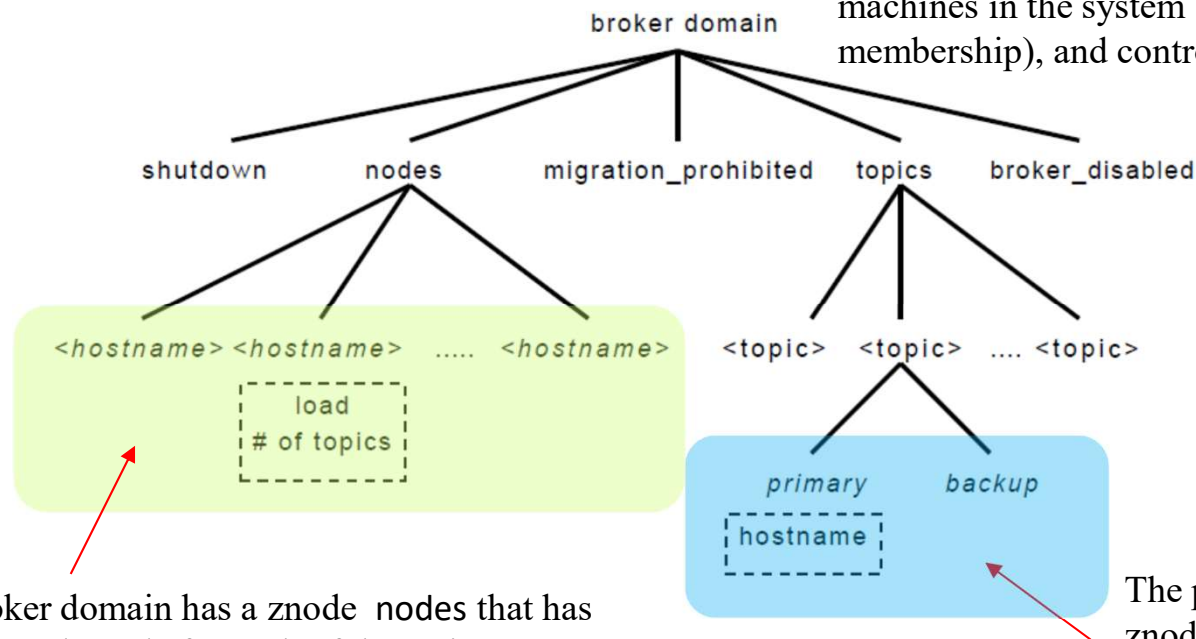
- Applications: enable clients to synchronize the beginning and the end of a computation.
 - Processes start their computation when enough processes, defined by the barrier threshold, have joined the barrier,
 - and leave the barrier once they have finished.
- Implementation using Zookeeper:
 - ... (figure out by yourself or see [paper](#))

❑ Leader Election

- ...

The layout of Yahoo! Message Broker (YMB, distributed publish-subscribe system) structures in ZooKeeper

YMB uses ZooKeeper to manage the distribution of topics (configuration metadata), deal with failures of machines in the system (failure detection and group membership), and control system operation.



Each broker domain has a znode `nodes` that has an ephemeral znode for each of the active servers that compose the YMB service.

Each YMB server creates an ephemeral znode under `nodes` with load and status information providing both group membership and status information through ZooKeeper.

The primary and backup server znodes not only allow servers to discover the servers in charge of a topic, but they also manage leader election and server crashes.

Source: [ZooKeeper: Wait-free Coordination for Internet-scale Systems](#). P. Hunt, et al. USENIX 2010.

See (<https://zookeeper.apache.org/doc/current/zookeeperUseCases.html>) and (<https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy>) for more use cases