APACHE HBASE

Session 5

Missing Piece in Core Hadoop

- Ability to access data randomly and close to real-time
- Not good for small files

Expectations ☺

- Data stored composed of much smaller entities, system transparently takes care of aggregating those files
- Some sort of indexing that allows user to retrieve data with minimal number of disk seeks
- Able to work with MapReduce

Apache HBase

- **Column-Oriented** data store, known as “Hadoop Database”
- Supports random real-time CRUD operations (unlike HDFS)
- Distributed – designed to serve large tables
- Open-source, written in Java
- Type of “NoSQL” DB **--** Does not provide a SQL based access
- Based on Google’s Big Table
- Horizontally scalable -- Automatic Shrading
- Strongly consistent reads and writes

HBase, is a sparse, distributed, persistent, multidimensional map, which is indexed by row key, column key, and a timestamp.

Can I Always use HBase??

- **Not suitable for every problem**
  - Compared to RDBMs has VERY simple and limited API
- **Good for large amounts of data**
  - 100s of millions or billions of rows
  - If data is too small all the records will end up on a single node leaving the rest of the cluster idle
- **Have to have enough hardware!!**
- **Two well-known use cases**
  - Lots and lots of data (already mentioned)
  - Large amount of clients/requests (usually cause a lot of data)
- **Great for single random selects and range scans by key**
- **Great for variable schema**
  - Rows may drastically differ
  - If your schema has many columns and most of them are null

Building Blocks

Table, Rows, Colums and Cells

- Most basic unit is **column**
- One or more column forms a **row** that is identified uniquely by **row key**
- A number of rows form a **table** and there can be many of them
- Each column may have **multiple versions**, with each version stored in separate **cell**
- All rows are always **sorted lexicographically** by their row-key
- **Row-key is always unique** which can be an arbitrary array of bytes

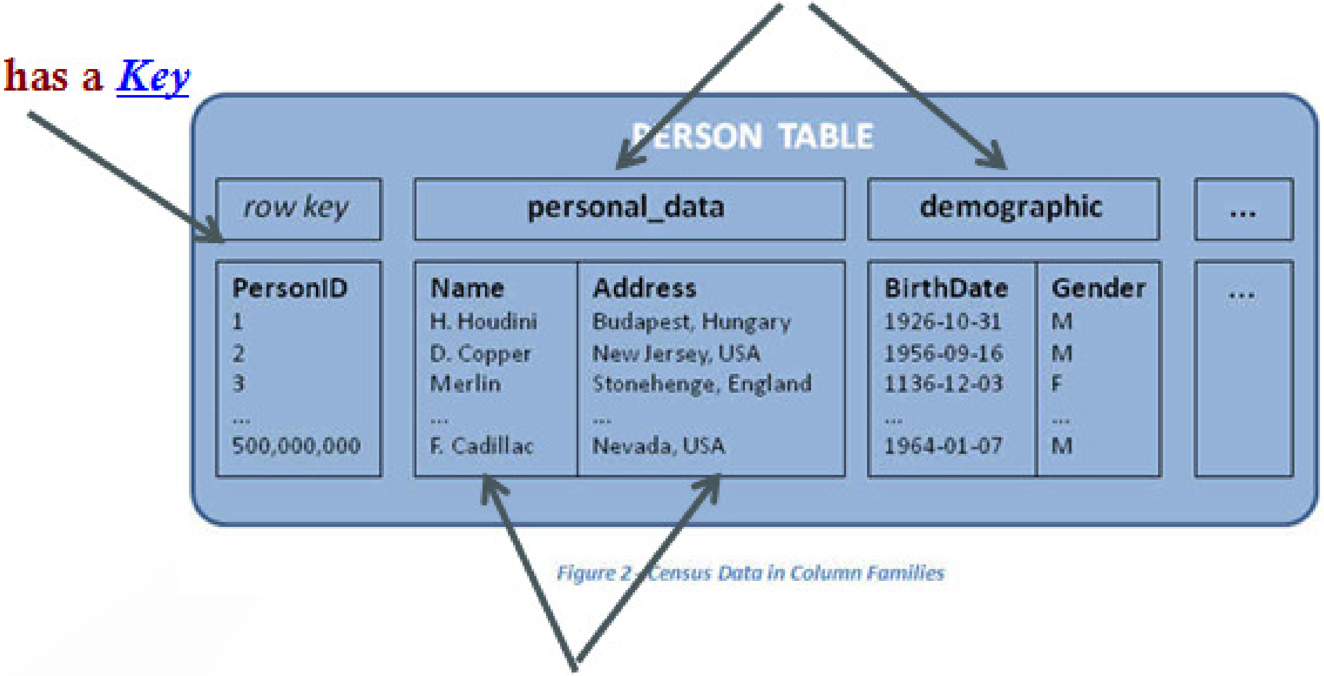
HBase Families

- Rows are composed of **columns**, those in turn grouped into **column-families**
- All columns in a column-family are stored together in same low level storage file called **HFile**.
- Name of column-family must composed of printable characters, a difference from others
- Columns are often referenced as **family:qualifier** with the qualifier being an arbitrary array of bytes
- **Storing NULL?** For Hbase, simply omit the whole column, i.e. NULLS are free of cost they do not occupy any space

HBase: Keys and Column Families

Each record is divided into *Column Families*

Each row has a *Key*



PERSON TABLE

row key	personal_data		demographic		...
PersonID	**Name**	**Address**	**BirthDate**	**Gender**	...
1	H. Houdini	Budapest, Hungary	1926-10-31	M	
2	D. Copper	New Jersey, USA	1956-09-16	M	
3	Merlin	Stonehenge, England	1136-12-03	F	
...	...	...	...	...	
500,000,000	F. Cadillac	Nevada, USA	1964-01-07	M	

Figure 2: Census Data in Column Families

Each column family consists of one or more *Columns*

HBase TimeStamps

- **Cells' values are versioned**
  - For each cell multiple versions are kept
    - 3 by default
- Another dimension to identify your data
- Either explicitly timestamped by region server or provided by the client
- Versions are stored in decreasing timestamp order
- Read the latest first – optimization to read the current value
- **You can specify how many versions are kept**

HBase Cells

- Can express access to data as ::

  (Table, RowKey, Family, Column, Timestamp) → Value

- Cells may exist in multiple versions, and different columns have been written in different times → API by default provides a coherent view, picking up the most current value for each cell

An Example

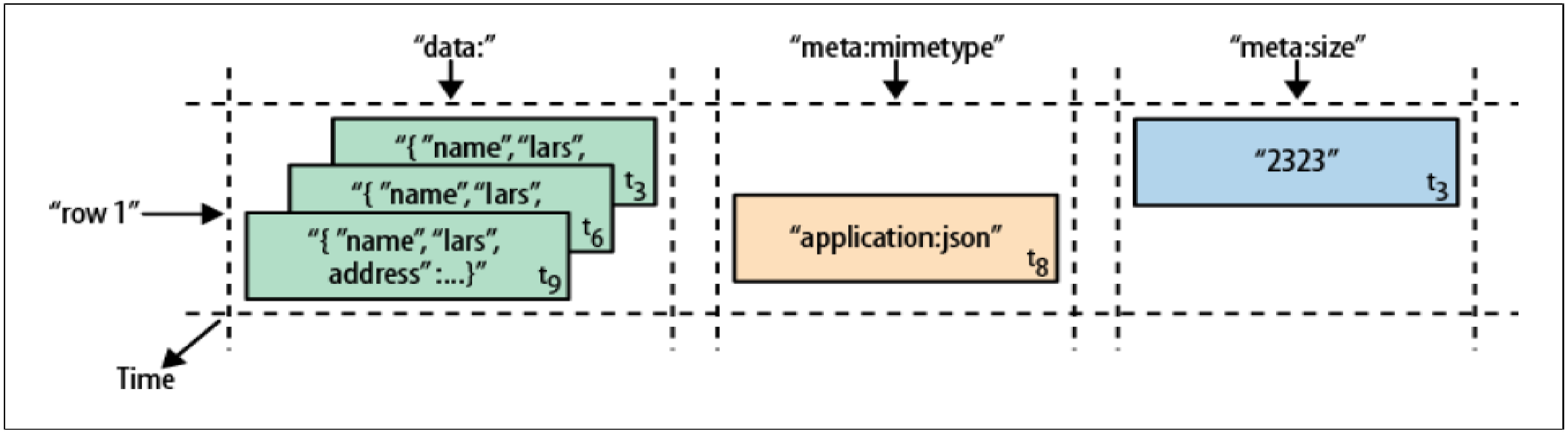
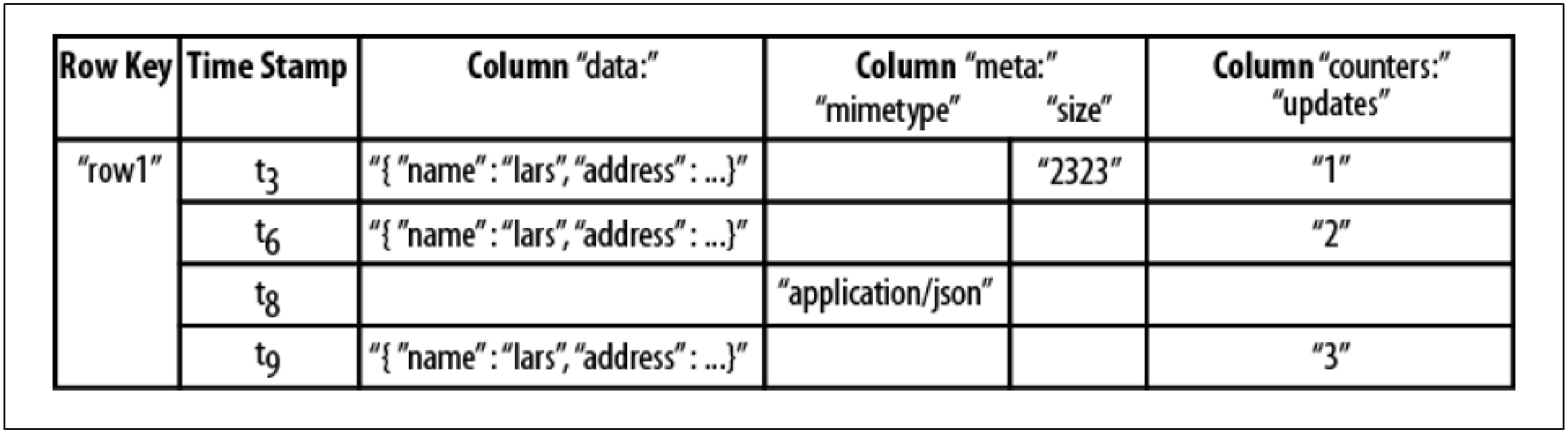


Figure 1-5. A time-oriented view into parts of a row



Row Key	Time Stamp	Column “data:”	Column “meta:”		Column “counters:”
			“mimetype”	“size”	“updates”
“row1”	t_3	“{ ”name” : “lars”, “address” : ...}”		“2323”	“1”
	t_6	“{ ”name” : “lars”, “address” : ...}”			“2”
	t_8		“application/json”		
	t_9	“{ ”name” : “lars”, “address” : ...}”			“3”

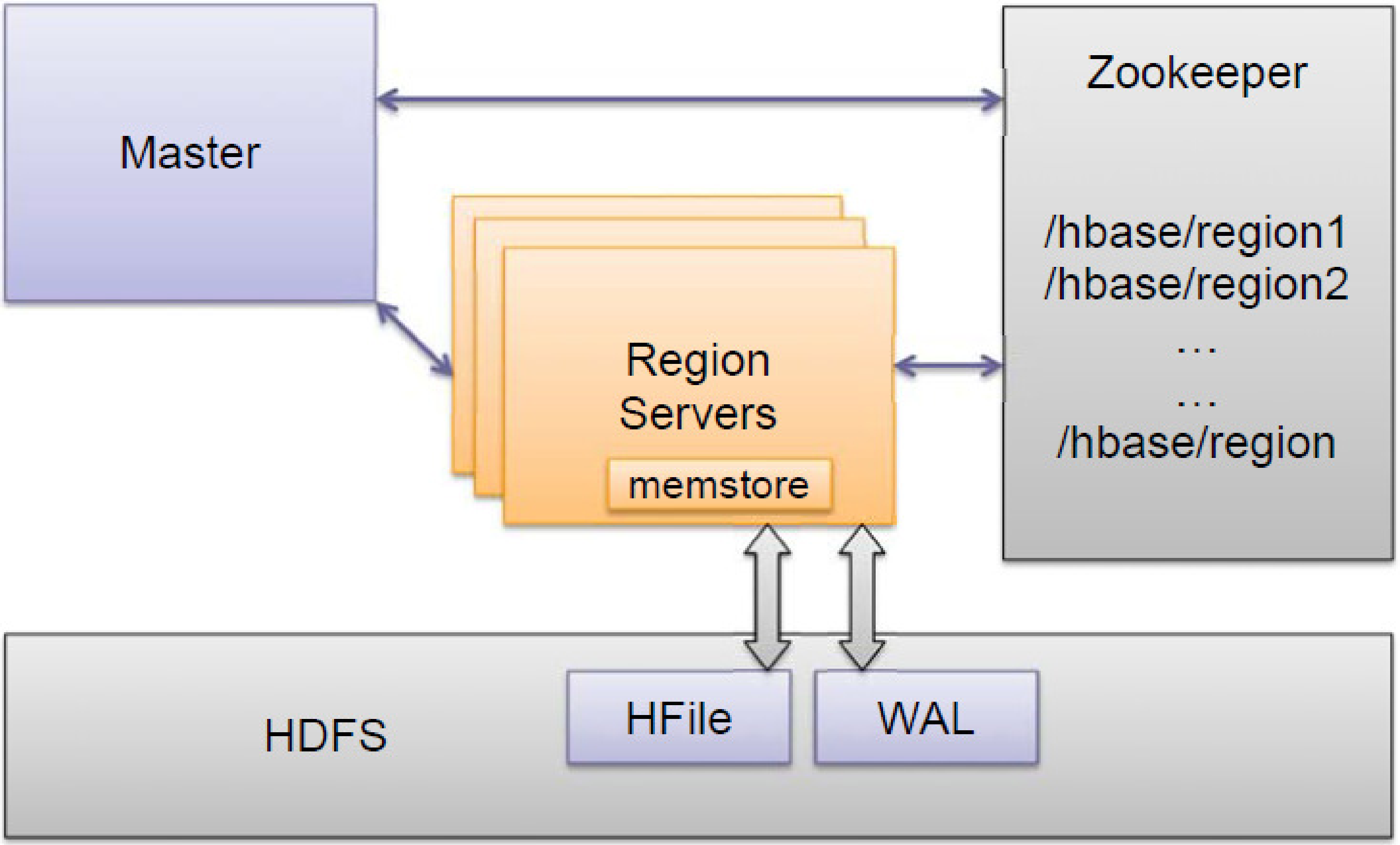
Figure 1-6. The same parts of the row rendered as a spreadsheet

HBase Architecture

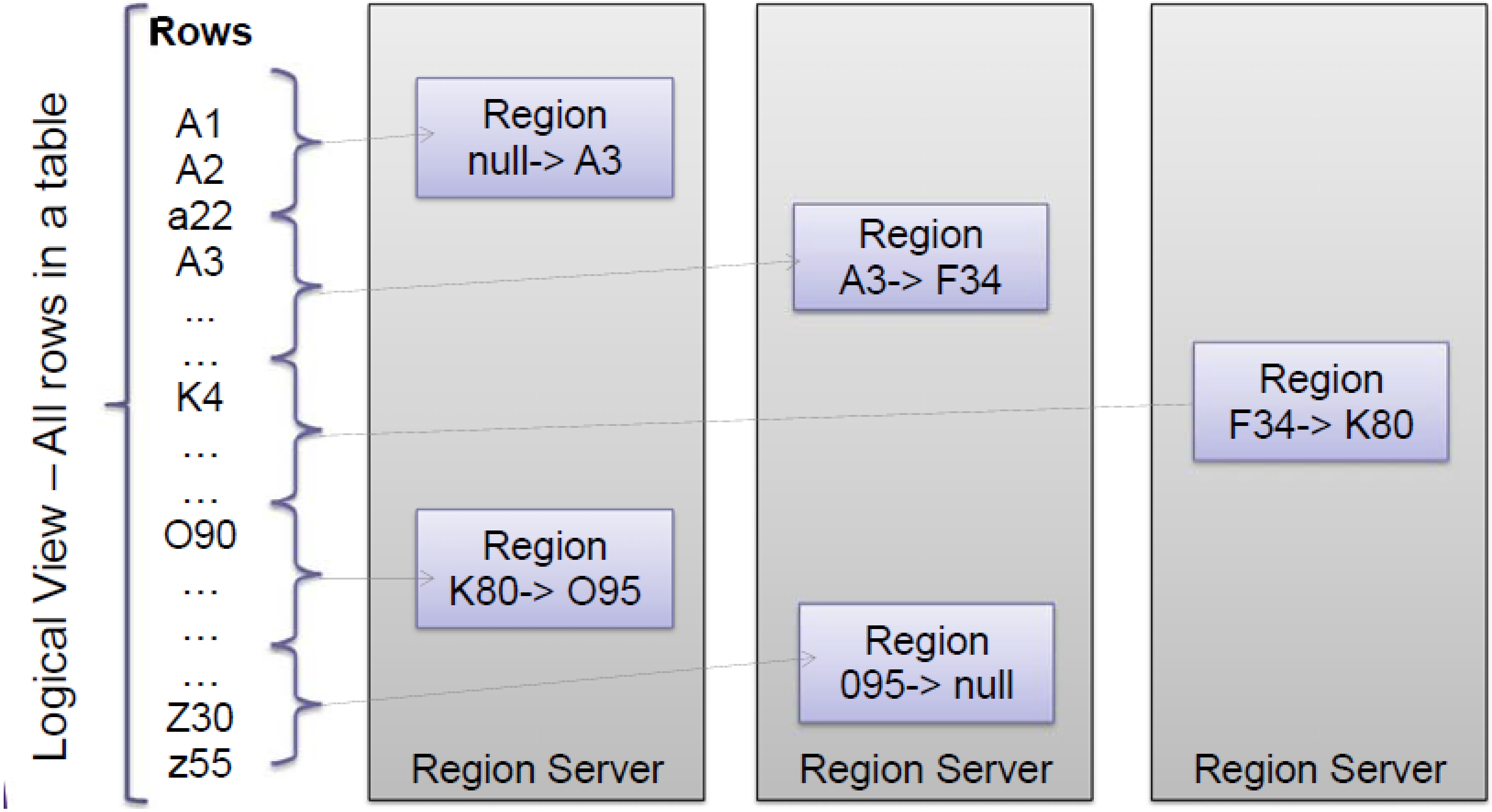
- **Table is made of regions**
- **Region – a range of rows stored together**
  - Single shard, used for scaling
  - Dynamically split as they become too big and merged if toosmall
- **Region Server- serves one or more regions**
  - A region is served by only 1 Region Server

- **Master Server –** daemon responsible for managing HBase cluster, aka Region Servers
- **HBase stores its data into HDFS**
  - relies on HDFS's high availability and fault-tolerance features
- **HBase utilizes Zookeeper for distributed coordination**

HBase Components



Rows Distribution b/w Region Servers



HBase Regions

- **Region is a range of keys**
  - start key → stop key (ex. k3cod → odiekd)
  - start key inclusive and stop key exclusive
- **Addition of data**
  - At first there is only 1 region
  - Addition of data will eventually exceed the configured maximum

    → the region is split

    - Default is 256MB
  - The region is split into 2 regions at the middle key
- **Regions per server depend on hardware specs, with today's hardware it's common to have:**
  - 10 to 1000 regions per Region Server
  - Managing as much as 1GB to 2 GB per region

Auto-Shrading

- Basic unit of scalability and load-balancing in Hbase is called a region.
  - Regions are essentially contiguous ranges of rows stored together.
- Regions are dynamically split by system when they become too large
- Each region is served by exactly one region-server, and each of these servers can serve many regions at a time
- **Regions allow for fast-recovery** when a server fails, and load balancing since they can be moved between servers

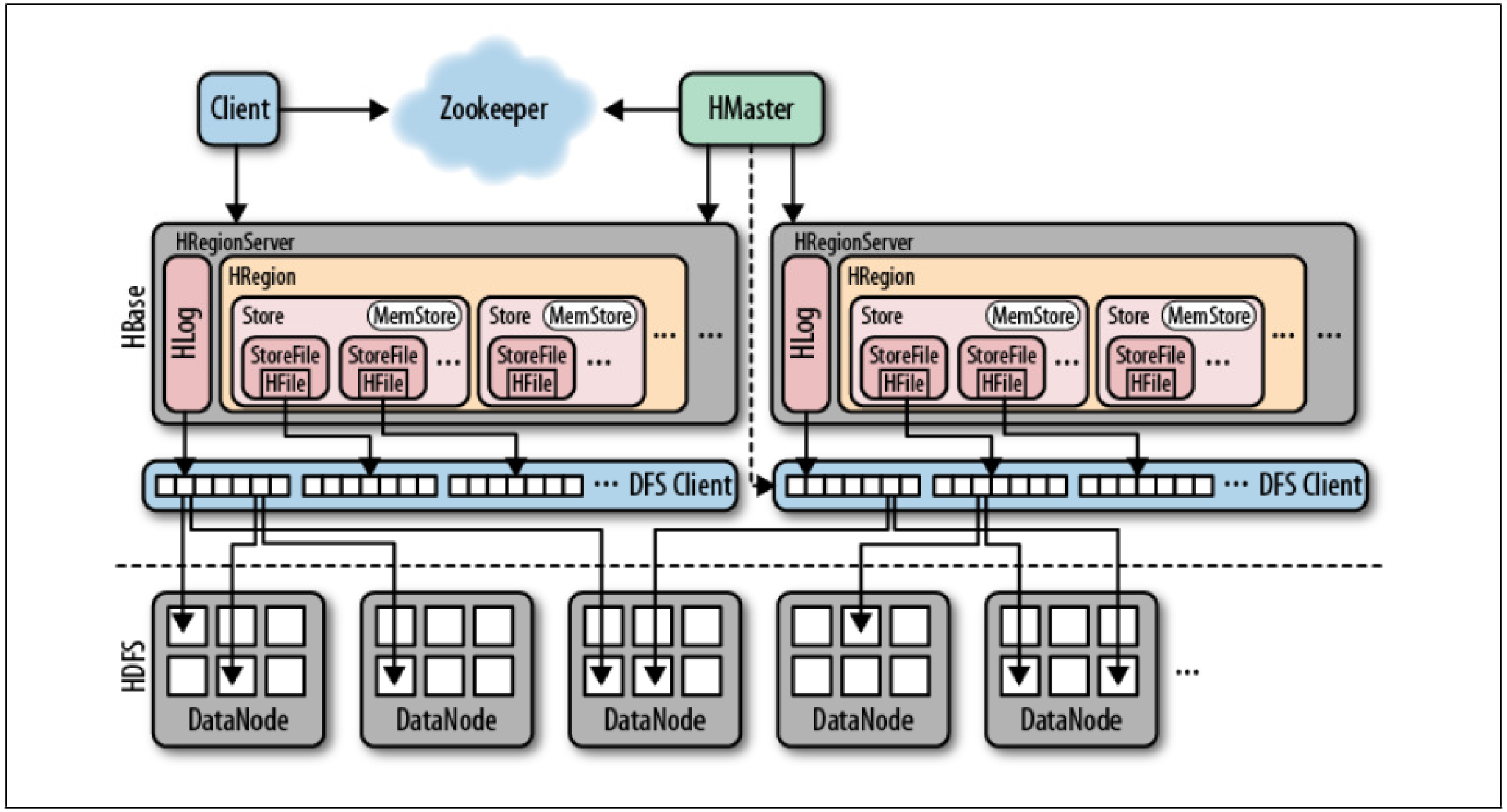
HBase Data Storage

- **Data is stored in files called HFiles/StoreFiles**
  - Usually saved in HDFS
- **HFile is basically a key-value map**
  - Keys are sorted lexicographically
- When data is added it's written to a log called Write Ahead Log (WAL) and is also stored in memory (memstore)
- **Flush:** when in-memory data exceeds maximum value it is flushed to an HFile
  - Data persisted to HFile can then be removed from WAL
  - Region Server continues serving read-writes during the flush operations, writing values to the WAL and memstore

HBase Data Storage

- Recall that HDFS doesn't support updates to an existing file therefore HFiles are immutable
  - Cannot remove key-values out of HFile(s)
  - Over time more and more HFiles are created
- **Delete marker is saved to indicate that a record was removed**
  - These markers are used to filter the data - to “hide” the deleted records
  - At runtime, data is merged between the content of the HFile and WAL
- Also supports **Predicate Deletions**
  - Allowing u to keep, for ex, only values written in past week

HBase Data Storage



Hbase basically handles two types of file types: one is used for WAL and other for actual data storage.

HFile Insight

- Internally, HFiles are sequences of blocks with block index stored at end of file
  - Default Block size is 64 KB but configurable
- Since, every Hfile has a block index, lookups can be performed with a single disk seek.
- First, the block possibly containing the given key is determined by doing a binary search in the in-memory block index, followed by a block read from disk to find the actual key.

Compaction

- **To control the number of HFiles and to keep cluster well balanced HBase periodically performs data compactions**

- **Minor Compaction:**
  - Smaller HFiles are merged into larger HFiles (n-way merge)
  - Fast - Data is already sorted within files
  - Delete markers are not applied
- **Major Compaction:**
  - For each region merges all the files within a column-family into a single file
  - Scan all the entries and apply all the deletes as necessary

HBase Master

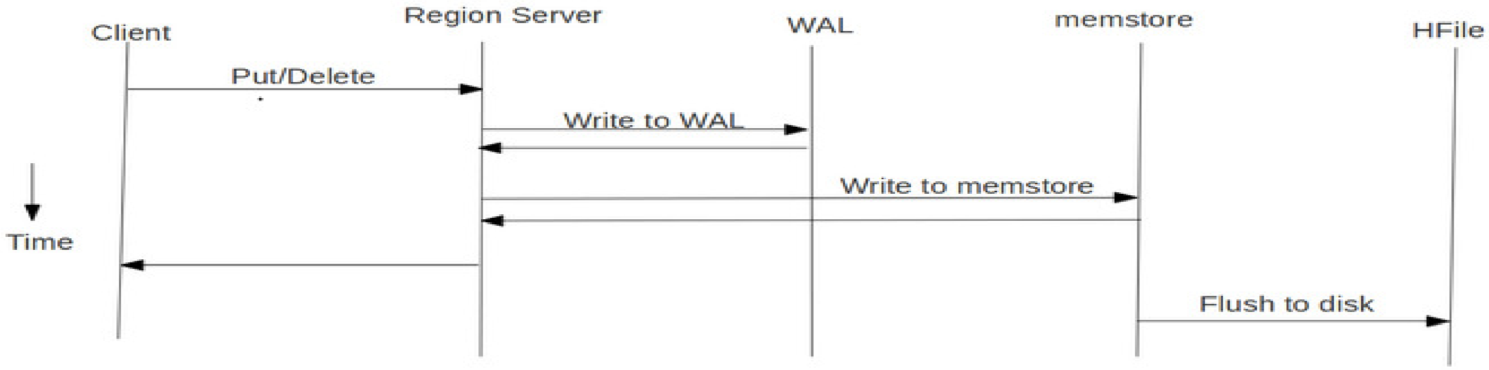
- **Responsible for managing regions and their locations**
  - Assigns regions to region servers
  - Re-balanced to accommodate workloads
  - Recovers if a region server becomes unavailable
  - Uses Zookeeper – distributed coordination service
- **Doesn't actually store or read data**
  - Clients communicate directly with Region Servers
  - Usually lightly loaded
- **Responsible for schema management and changes**
  - Adding/Removing tables and column families

HBase and Zookeeper

- **Each Region Server creates an ephemeral node**
  - Master monitors these nodes to discover available region servers
  - Master also tracks these nodes for server failures
- **Uses Zookeeper to make sure that only 1 master is registered**
- **HBase cannot exist without Zookeeper**

HBase Write Path

- Client issues a Put request to HRegionServer, which hands the details to matching HRegion instance
- First step is to write data to Write-Ahead-Log (WAL)
- Once data is written to WAL, it is placed in memstore
- At same time, it is checked to see if memstore is full and, if so, a flush to disk is requested



HBase Write Path

HBase Read path

- Reading data back involves a merge of what is stored in the memstores, that is, the data that has not been written to disk, and the on-disk store files.

- **Communication Flow to Access a Row:**
  - New client contacts the Zookeeper ensemble to retrieve the servername that hosts the -ROOT- region
  - It then query that region server to get server name that hosts .META. Table region containing the required row
  - Both of these information is cached and lookup only once
  - Lastly, it query the reported .META. server to retrieve the server name that has the region containing the row key the client is looking for

HBase Read Path Contd…

- – Client caches this information as well and then contacts HRegionServer hosting that region directly
- – Overtime client has pretty complete picture of where to get rows without needing to query .META. server again.

- Note that the **WAL is never used during data retrieval**, but solely for recovery purposes when a server has crashed before writing the in-memory data to disk.

HBase Lab Session

Planned Contents –

- ✓ Start the HBase server and launch the HBase shell
- ✓ Create a table and populate it with data
- ✓ Learn how to check the health of HBase
- ✓ View the HBase web GUI
- ✓ Track down the HBase files in HDFS

Thank You