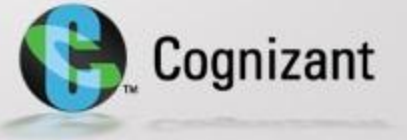
KEEP CHALLENGING™



HBASE INTRODUCTION

What is HBase?

- HBase is an open source, non-relational, distributed database modeled after Google's BigTable and written in Java.

- Hbase is also ...
  - A Sorted Map
  - Open/source
  - Sparse
  - Multidimensional
  - Runs on top of HDFS
  - Modeled after Google’s BigTable

- Think of it as a sparse, consistent, distributed, multidimensional, sorted map:
  - labeled tables of rows
  - row consist of key-value cells:

  (row key, column family, column, timestamp) -> value

- Hbase is a Map at its core
  - Much like a PHP/Perl associative array, or JavaScript Object
  - The map is indexed by a row key, column key, and a timestamp
  - Each value in the map is an uninterpreted array of bytes
- Sorted
  - Key/Value pairs are kept in lexicographic sorted order
  - Very important when scanning large amounts of data

- Ensures like information is located in close proximity
- Impacts row/key design considerations

- Distributed
  - Built upon a distributed filesystem(HDFS)
  - Underlying file storage abstracts away complexities of distributed computing

- Sparse
  - A given row can have any number of columns
  - Not all columns must have values
  - NULLs are not stored
  - May be gaps between keys

- Multidimensional
  - All data is versioned using a timestamp (or configurable integer)
  - Data is not updated in place
  - Instead, a new version is added with a later version number

What is HBase?

- HBase is a type of "NoSQL" database.

- "NoSQL" is a general term meaning that the database isn't an RDBMS which supports SQL as its primary access language, but there are many types of NoSQL databases:

1. Key-Value Store – It has a Big Hash Table of keys & values {Example- Riak, Amazon S3 (Dynamo)}
2. Document-based Store - It stores documents made up of tagged elements. {Example- CouchDB}
3. Column-based Store - Each storage block contains data from only one column, {Example- HBase, Cassandra}
4. Graph-based-A network database that uses edges and nodes to represent and store data. {Example- Neo4J}

HBase is NOT a Traditional RDBMS

	HBase	RDBMS
Data Layout	Column Family-oriented	Row or Column-oriented
Transactions	Single row only	Yes
Query language	Get/Put/Scan	SQL
Security	Kerberos	Authentication/Authorization
Indexes	Row-Key only	Yes
Max. Data Size	PB+	TBs
Read/Write throughput limits	Millions of queries/second	1000s queries/second

RDBMS vs HBase

HBase	RDBMS
Flexible schema, add columns on the fly	Fixed schema
Good with sparse tables	Not optimized for sparse tables
Wide tables	Narrow tables
Joins using MR –not optimized	Optimized for joins (small, fast ones too!)
Tight integration with MR	Not really...
De-normalize your data	Normalize as you can
Horizontal scalability –just add hardware	Hard to shard and scale
Consistent	Consistent
No transactions	Transactional
Good for semi-structured data as well as structured data	Good for structured data

HBase is built on Hadoop

- Hadoop Provides :
  - Fault Tolerance
  - Scalability
  - Batch Processing with MapReduce

- HBase Provides :
  - Random reads and writes
  - High throughput
  - Caching

- Lots of data
  - Hundreds of Gigabytes up to Petabytes
- High write throughput
  - 1000s/second per node
- Scalable cache capacity
  - Adding nodes adds to available cache
- Data layout
  - Excels at key lookup
  - No penalty for sparse columns
- Rows from an HBase table can be used as input to a MapReduce job
  - Each row is treated as a single record
  - MapReduce jobs can sort/search/index/query data in bulk

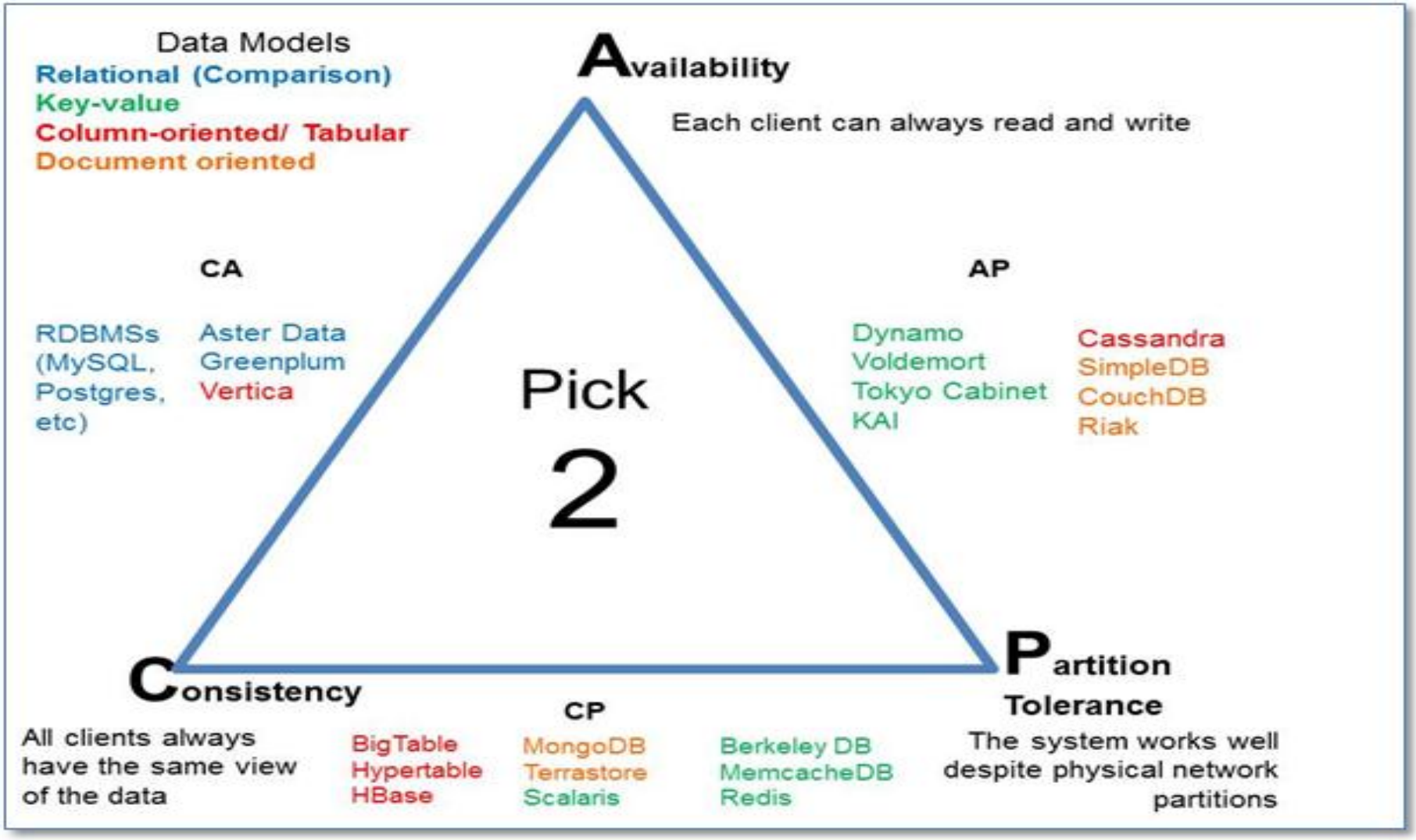
HBase Use Case

- WorldLingo
  - Hardware: 44 servers (each server has: 2 dual core CPUs, 2TB storage, 12GB RAM)
  - Two separate Hadoop/HBase clusters with 22 nodes each.
  - Hadoop is primarily used to run Hbase and MapReduce jobs scanning over the HBase tables to perform specific tasks.
  - Hbase is used as a scalable and fast storage back end for millions of documents.
  - Stores 12 million documents with a target of 450 million in the near future.
- eBay Cassini search-engine rewrite project
- HBase in production
  - Facebook, StumbleUpon, TrendMicro, Ning, and many more: http://wiki.apache.org/hadoop/Hbase/PoweredBy

When to use HBase

- Use HBase if…
  - You need random write, random read, or both (but not neither)
  - You need to do many thousands of operations per second on multiple TB of data
  - Your access patterns are well/known and simple
- Don’t use HBase if…%
  - You only append to your dataset, and tend to read the whole thing
  - You primarily do ad/hoc analytics (ill/defined access patterns)
  - Your data easily fits on one beefy node

What is CAP Theorem?



Why Use HBase?

- Storing large amounts of data(TB/PB)
- Storing unstructured or variable column data
- Big Data with random read and writes
- Horizontally scalable
    - Automatic sharding
- Strongly consistent reads and writes
- Simple Java API
- Integration with Map/Reduce framework

- Sharding :
  – sharding describes the logical separation of records into horizontal partitions.
  – The idea is to spread data across multiple storage files—or servers—as opposed to having each stored contiguously.
  – The separation of values into those partitions is performed on fixed boundaries: you have to set fixed rules ahead of time to route values to their appropriate store. With it comes the inherent difficulty of having to reshard the data when one of the horizontal partitions exceeds its capacity.

What HBASE is NOT

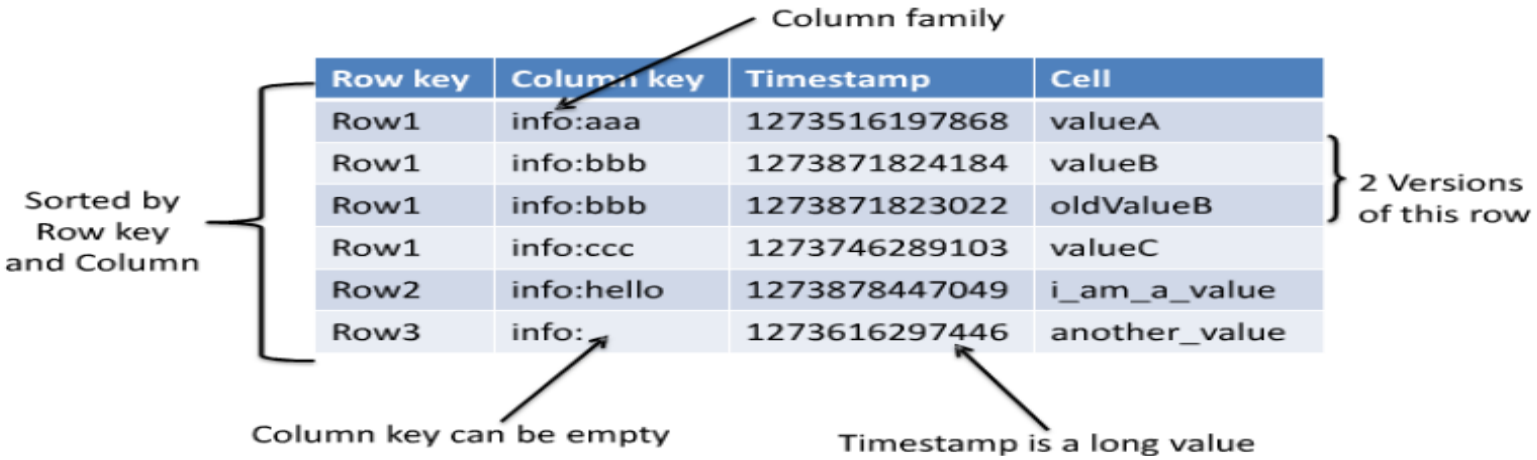
- Not an SQL database
- Not relational
- No joins
- No fancy query language and no sophisticated query engine
- No transactions out-of-the box
- No secondary indices out-of-the box
- Not a drop-in replacement for your RDBMS

HBASE Data Model

- Tables are comprised of rows and columns
- Every row has a row key (analogous to a primary key)
  - Rows are stored by row key for fast lookups
- All Columns in HBase belong to a particular column family
- A table may have 1 or more column families
  - Common to have a small number of column families
  - Column families should rarely change
  - A column family can have number of columns
  - Columns within a family are sorted and stored together
  - Columns only exist when inserted
    - Nulls are free
- Each row has a timestamp
  - Multiple versions of a row can exist

HBASE Data Model

- Column consists of column family prefix + qualifier
- Separate column families are useful for
  - Data that is not frequently accessed together
  - Data that uses different column family options ex: compression
- Table is a distributed sorted map
  - **row key + column family + column + timestamp -> value**



Column Family Attributes

- COMPRESSION
  - NONE, GZ, LZO
  - By Default: NONE
- VERSIONS
  - 1+
  - By Default: 3
- TTL
  - 1-2147483647
  - By Default: 2147483647
- BLOCKSIZE
  - 1byte – 2GB
  - By Default: 64k
- IN_MEMORY
  - True, False
  - By Default: False
- BLOCKCACHE
  - True, False
  - By Default: True

Versions of Data

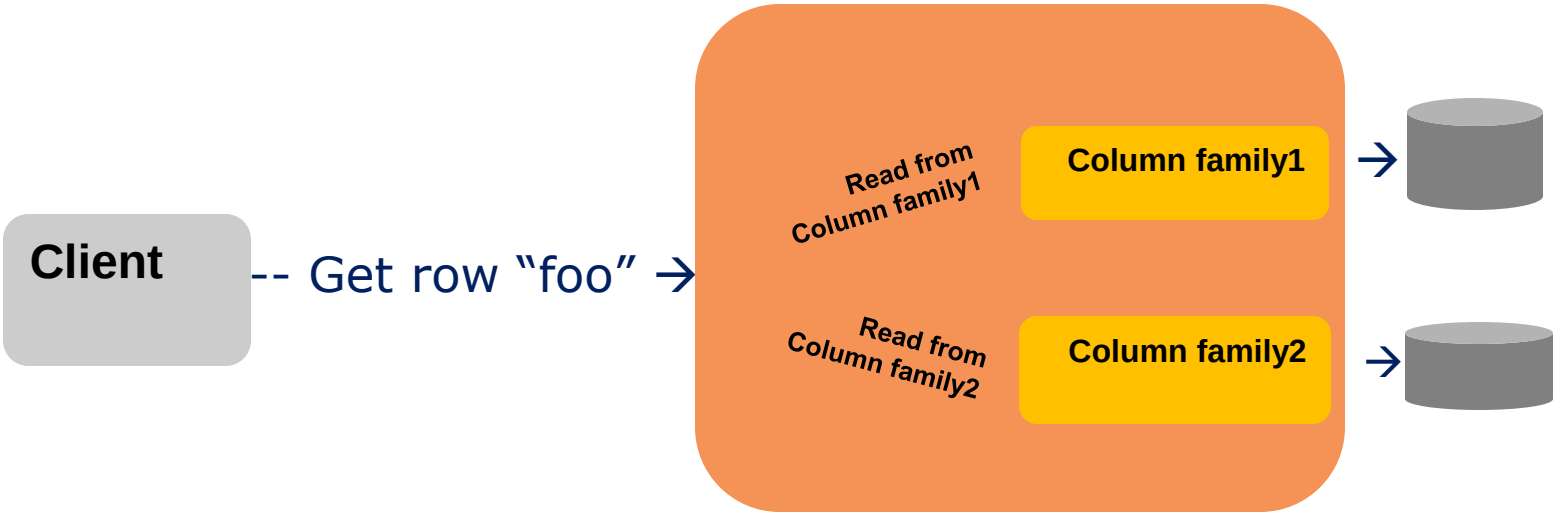
- Keeps 3 versions of a row
- Sorted by their timestamp (descending order)

Key	Column	Value	Timestamp
rowA	Fam:foo	New value	1275340679713
rowA	Fam:foo	Old value	1275091706190
rowB	Fam:foo	Some value	1274999316683

Sorted in descending order

Designing Tables

- Same row key → same node
  - Lookups by row key talk to a single node or "region server"
- Same Column family → same set of physical files
  - Retrieving data from a column family is sequential I/O



Comparison with RDBMS design

- Scaling relational tables often means partitioning or sharding data
  - HBase automatically partitions data in regions
- In relational databases, one might normalize tables and use joins to retrieve data
  - HBase does not support explicit joins
  - A lookup by row key implicitly joins data from column families if necessary

HBase Columns and Column Families

- Columns are grouped into Column Families (CFs)
- All column family members have the same prefix
  - Ex: metadata:fname and metadata:lname
  - The ":" delimits the CF from the qualifier
- Columns can be created on the fly
- Physically, all column family members are stored together
- Column families must be declared at schema definition time
- Tuning and storage settings can be specified for each column family

-

Attribute	Possible values	Default
COMPRESSION	NONE,GZ,LZO, SNAPPY	NONE
VERSIONS	1+	3
TTL	1-2147483647 (seconds)	FOREVER(special value means the data is never deleted)
BLOCKSIZE	1byte – 2GB	64K
IN_MEMORY	true, false	false
BLOCK CACHE	True, false	false

Column Family Attributes

- In-Memory
  - column families can be defined as in-memory
  - Data is still persisted to disk
  - In-memory blocks have the highest priority in the Block cache
  - Not guaranteed that the entire table will be in memory

- Time To Live (TTL)
  - Column families can set a TTL length in seconds
  - Rows will automatically be deleted when the expiration is reached
  - Applies to all versions of a row
  - sets by HColumnDescriptor
  - Used in conjunction with minimum versions setting

Number of Versions

- Row values are never overwritten
  - Stores different values per row by time
  - Excess versions are physically removed during a background file cleanup process called compaction

- Number of versions is configured per column family default is 3
- The larger the number of versions, the larger the store file size
- Minimum number of versions default = 0
  - Feature is disabled by default
  - Used with TTL parameter on a column family

HBase Cells and Versions

- Table cells are the intersection of a row and column
- A {table, rowkey, column_family, column, timestamp} tuple specifies a cell in HBase
- Cell content contains uniterpreted bytes
- Cells are versioned
  - unlimited number of versions in a cell
  - version is specified using a long integer and stored in decreading order

Supported Data types

- Bytes-in/out interface
- Anything that can be converted to an array of bytes can be stored
- Cell size
- Counters synchronization is done on Region server (not client)

HBase Operations

- Get/Scan retrieve data
  - By default, cell with largest value for version is returned
  - *Get.SetMaxVersions()* to return more than 1 version
  - *Get.SetTimeRange() to return versions other than the latest*

- Put inserts data
  - doesn't replace an existing cell i.e.always create a new version of cell
  - Versions can be specified on a per-column basis and default version is currentTimeMillis
- Delete marks data being deleted
  - data is never modified in place and deleted data isn't removed immediately
  - Delete creates a 'Tombstone marker' which masks deleted values
  - Data is removed during background file cleanup process called major compaction

HBase shell and useful commands

Entering a HBase shell
  – $ hbase shell
Help – lists all the shell commands
Status – shows basic status about the cluster
List – lists all user tables in HBase
Describe ‘<tablename>’ – returns structure of table

Creating Tables
- General form
  – create ‘<tablename>’, {NAME=>’<colfam>’ [, <options>] } [,{...}]

Create Command

- create a table in HBase shell stores information
  - column families are created at schema definition time
- create 't1', {NAME => 'cf1', VERSIONS => 5}
- create 't1', {NAME => 'cf1'}, {NAME => 'cf2'}, {NAME => 'cf3'}
- create 't1', 'cf1', 'cf2', 'cf3'
- create 't1', {NAME => 'cf1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}

Get & Put Command

- Retrieving rows using get command
- get ‘tablename’, ‘rowkey’ [,options]
- get 't1', 'r1'
- get 't1', 'r1', {TIMERANGE => [ts1, ts2]}
- get 't1', 'r1', {COLUMN => 'c1'}
- get 't1', 'r1', {COLUMN => ['c1', 'c2', 'c3']}
- Inserting rows with put command
- put ‘tablename’, ‘rowkey’, ‘colfam:col’, ‘value’ [, timestamp]
- put 't1', 'r1', 'c1', 'value', ts1

Scan Command

- Scan command will retrieve multiple rows
- scan ‘tablename’, [,{options}]
- may include options for COLUMNS, STARTROW, STOPROW, TIMESTAMP, LIMIT
- scan 't1'
- scan 't1', {COLUMNS => ''}
- scan 't1', {COLUMNS => ['c1', 'c2'], LIMIT => 10, STARTROW => 'xyz'}
- scan 't1', {COLUMNS => 'c1', TIMERANGE => [1303668804, 1303668904]}
- scan 't1', {COLUMNS => ['c1', 'c2'], CACHE_BLOCKS => false}

Count Command

- Count command is used to find number of rows in a table
- count ‘tablename’, [, interval]
- counting the rows of large table can be slow
- if specified, the progress will be reported every interval rows
- count 't1'
- count 't1', INTERVAL => 100000
- count 't1', CACHE => 1000
- count 't1', INTERVAL => 10, CACHE => 1000

Removing Data

Delete columns from a row
- delete ‘tablename’, ‘rowkey’, ‘col’

To delete an entire row
- deleteall ‘tablename’, ‘rowkey’

Delete all rows
- truncate ‘tablename’

Drop a table
- drop ‘tablename’

Changing Column Families

To change column families table must be disabled first
- disable ‘tablename’

Add, change or remove column families
- alter ‘tablename’, {NAME =>’colfam’, [,options] }

Re-enable the table – enable ‘tablename’

Help Tools

lists the various advanced commands like

1. major_compact : cause a major compaction of table
2. flush : cause region servers to flush memstore for a table
3. split : cause a table to split each region

Scripting

passing scripts to HBase shell

${HBASE_HOME}/bin/hbase shell PATH_TO_SCRIPT

pipe commands to the bin/hbase shell command

$ echo "list" | bin/hbase shell | ./filter_table_names.pl > table_names.txt

The above script uses to create filtered list of table names and then uses table_names.txt file as a input to a script

$ ./turn_table_names_into_disable_delete_command.pl table_names.txt | bin/hbase shell

will deletes those tables.

HBASE Daemons

- **HMaster**
  - Responsible for coordinating the slaves (HRegionServer)
  - Assigns regions, detects failures of HRegionServers and controls some admin functions
- **Catalog Tables**
  - Keep track of locations of region Servers and region
- **HRegionServer**
  - Serves data for reads and writes
- **Region**
  - A set of rows belonging to a table
- **HQuorumPeer/Zookeeper**
  - It is known as ZooKeeper.
  - In HBase, ZooKeeper coordinates, communicates, and shares state between the Masters and RegionServers.

Zookeeper service

- Stores global information about cluster
- provides synchronization and detects master node failure
- holds the location of Root table and master

HBasemaster

- Responsible for co-ordinating region servers
- Assigns regions, detects region server failures
- Handles schema design
- Master runs several background threads
  - LoadBalancer periodically reassigns regions in the cluster
  - CatalogJanitor periodically checks and cleans up .META table
- All HBase cluster can have multiple masters
  - upon startup all complete to run cluster

Few HBase Terminology

Table	(HBase table)
Region	(Regions for the table)
Store	(Store per ColumnFamily for each Region for the table)
MemStore	(MemStore for each Store for each Region for the table)
StoreFile	(StoreFiles for each Store for each Region for the table)
Block	(Blocks within a StoreFile within a Store for each Region for the table)

Region Servers

- serve data for reads and writes of row contained in regions
- will split a region that has too large
- runs several background threads

1. CompactSplitThread checks for splits and handle minor compactions
2. MajorCompactionChecker checks for major compactions
3. MemStoreFlusher periodically flushes in-memory writes in the memstore to store files
4. LogRoller periodically checks region servers WAL

Catalog Tables

- ROOT- catalog table : which lists location of .META table

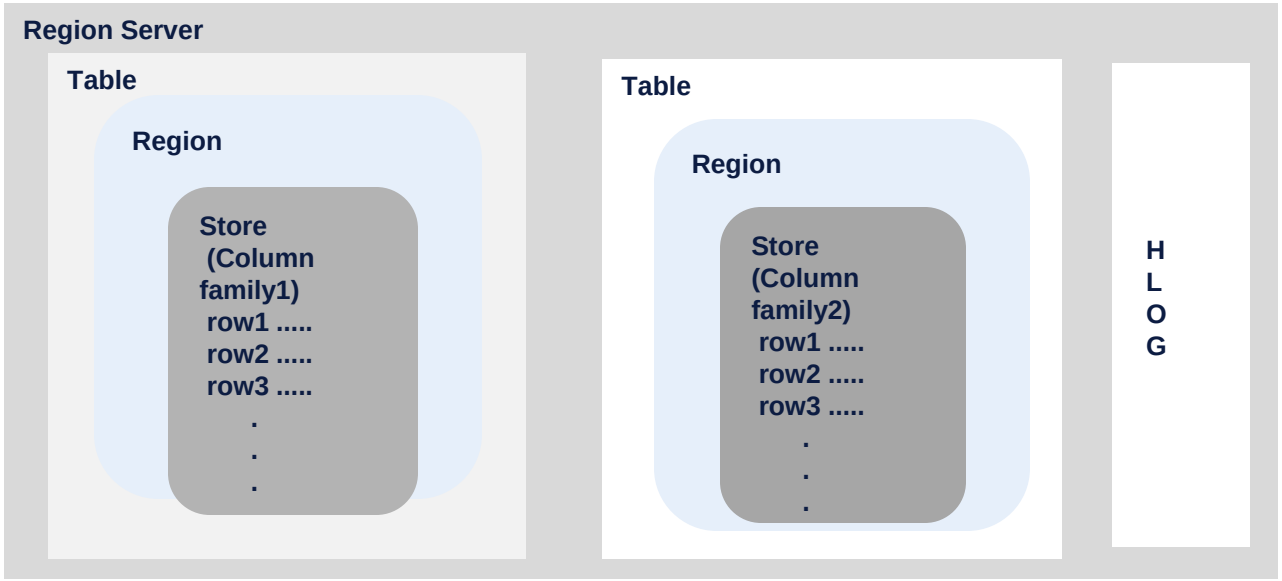
ROW	COLUMN+CELL
.META.,,1	c o l u m n = i n f o : regioninfo, timestamp=1309921036780, value=REGION => {NAME => '.META.,,1', STARTKEY => '', END KEY => '', ENCODED => 1028785192, TABLE => {{NAME => '.META.', IS_META => 'true', FAMILIES => [{NAME =>'info', BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSIONS => '10', COMPRESSION => 'NONE', TTL=> '2147483647', BLOCKSIZE => '8192', IN_MEMORY => 'true', BLOCKCACHE => 'true'}]}} .META.,,1

Regions

– Holds a subset of tables rows like a partition i.e, region is specified by startKey and endKey

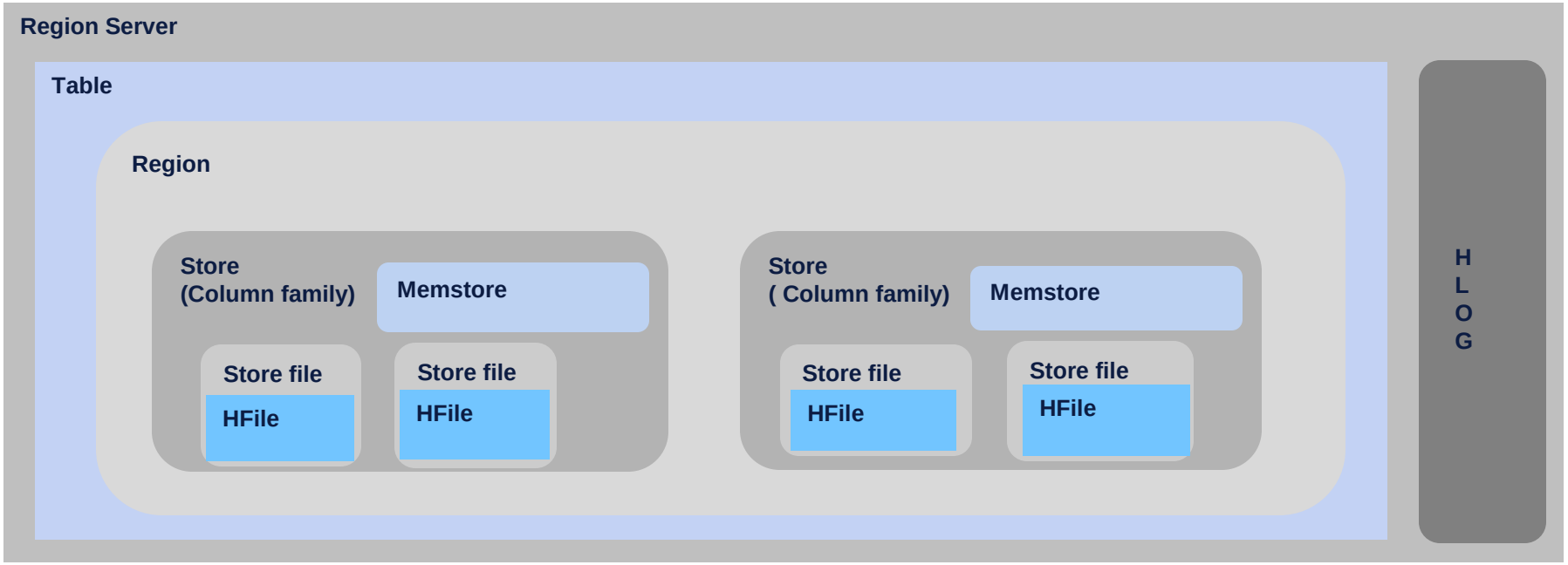
– table may have one or more regions comprises of store per col.family

– New regions are automatically created as tables grow i.e, region may live on different node and made of several store files (HDFS files)

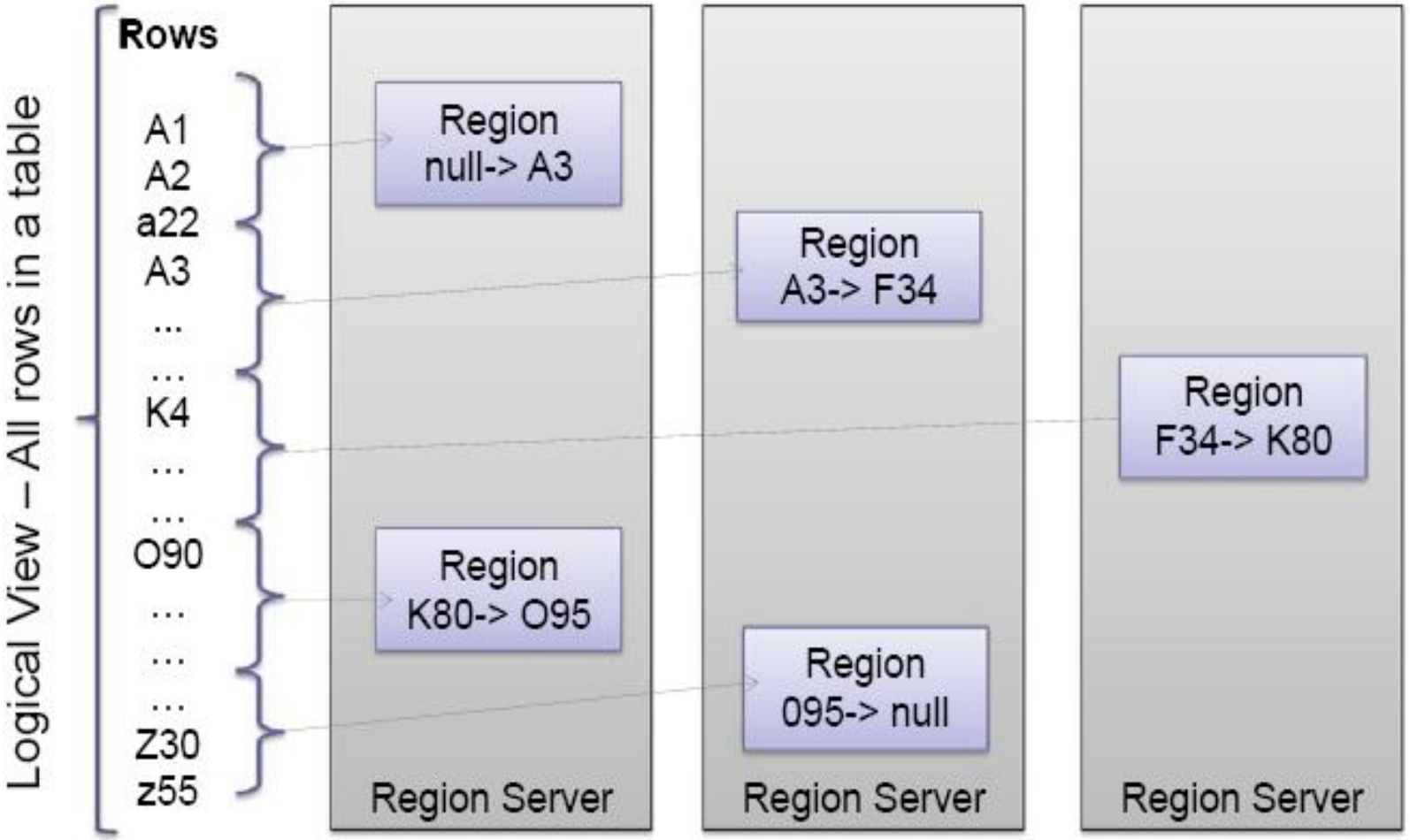


Region Store

- Store hosts a memstore and 0 or more store files
- corresponds to a column family for a table and for a given region
- Memstore holds in-memory modifications to store and flush writes to disk and clears memstore
- Store file is a actual data storage and stored in HFile format



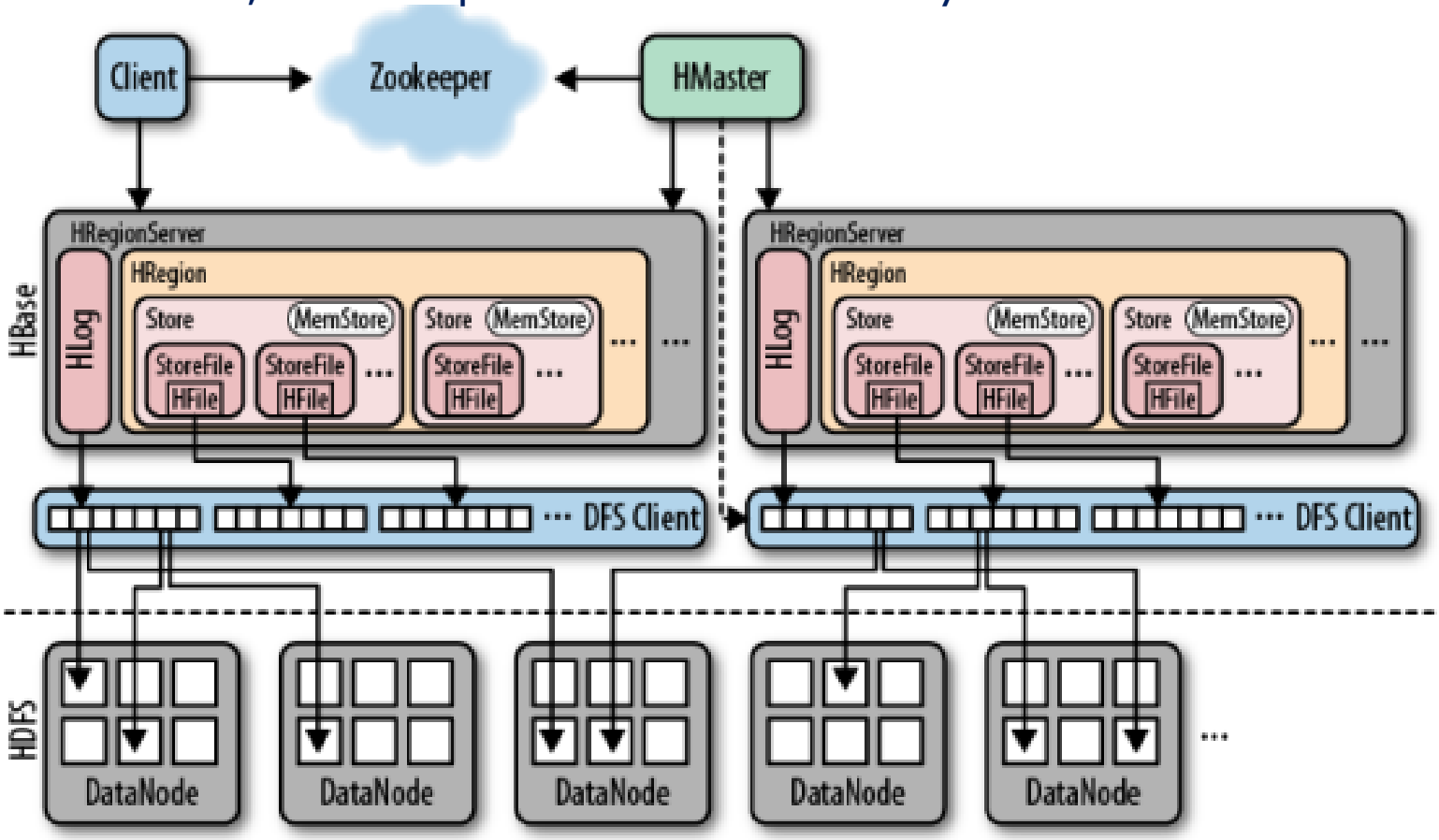
Rows Distribution Between Region Servers



Cognizant

HBase High Level Architecture

Region Files and HDFS : Files are divided into smaller blocks when stored in HDFS, block replication is handled by HDFS



Cognizant

Region Server Data Locality

- HDFS Client writes 3 replicas of each block by default
  - 1st replica is written to local node
  - 2nd and 3rd replicas written to 2 other nodes in a different rack
- Region splits and re-assignments can move data to non-local store files
- HBase eventually achieves locality for a region after a flush or major compaction

Region Splits

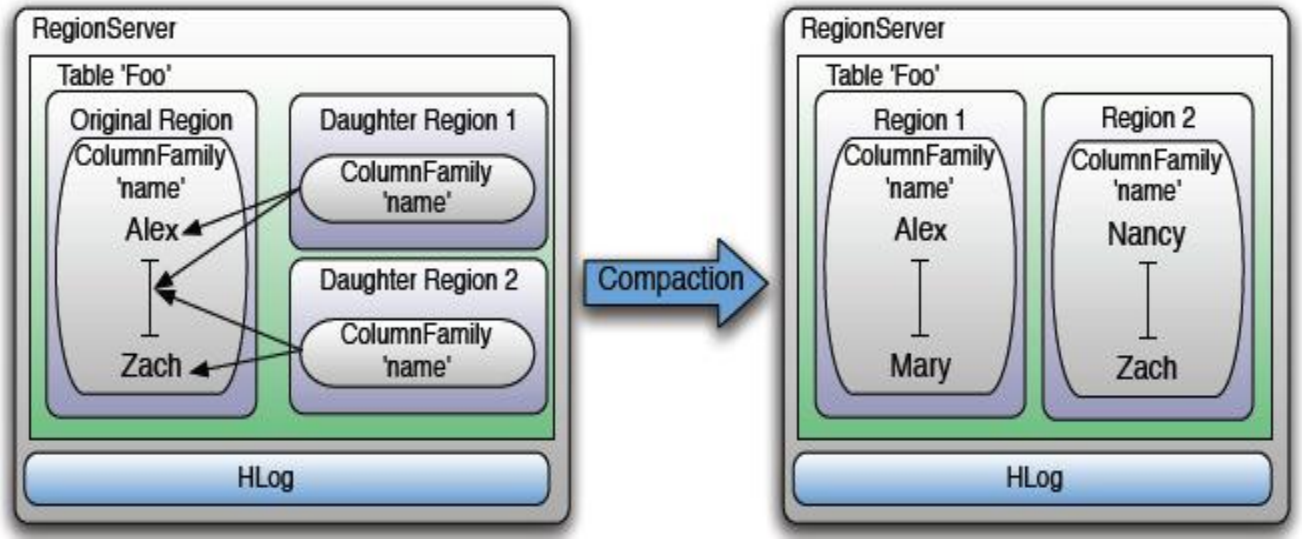
- When regions get too big (128MB by default) they are automatically split – HBase creates 2 daughter reference files contains key where region was split

– Major compaction original data files rewritten to separate files in new region directory, small reference files and original region are removed

- size at which a region splits is controlled by hbase.hregion.max.filesize property

- Region Server handles the region splits

  – off-lines the original region
  – adds daughter regions to .META
  – opens daughter regions on parent region server, and it reference original regions until compaction.
  – reports the splits back to HMaster



- Region Size
  - Basic elements of availability and distribution i.e. We don't want high number of regions for small amount of data
  - High region count (e.g.3000) can impact performance
  - Low region count prevents parallel scalability
  - A lower number of regions is preferred, generally in the range of 20 to low-hundreds of region server

- Load Balancer
  - Automatically moves regions around to balance cluster load
  - period when this runs is configurable

- Data Distribution – distributed across regions based on row keys

Data Storage

- Data is first written to the region's Write-Ahead-Log(WAL) and then to memstore
  - The WAL is required for crash recovery if the memstore is lost
- Memstore is flushed to an immutable file (store file) when one of the following occurs:
  - hbase.regionserver.global.memstore.upperLimit is reached
  - hbase.hregion.memstore.flush.size is reached
  - hbase.hregion.preclose.flush.size is reached and the region is being closed
- Flushes can block updates
  - When hbase.hregion.memstore.block.multiplier* hbase.hregion.flush.size is reached
- Eventually these store files will be aggregated and cleaned up during a compaction

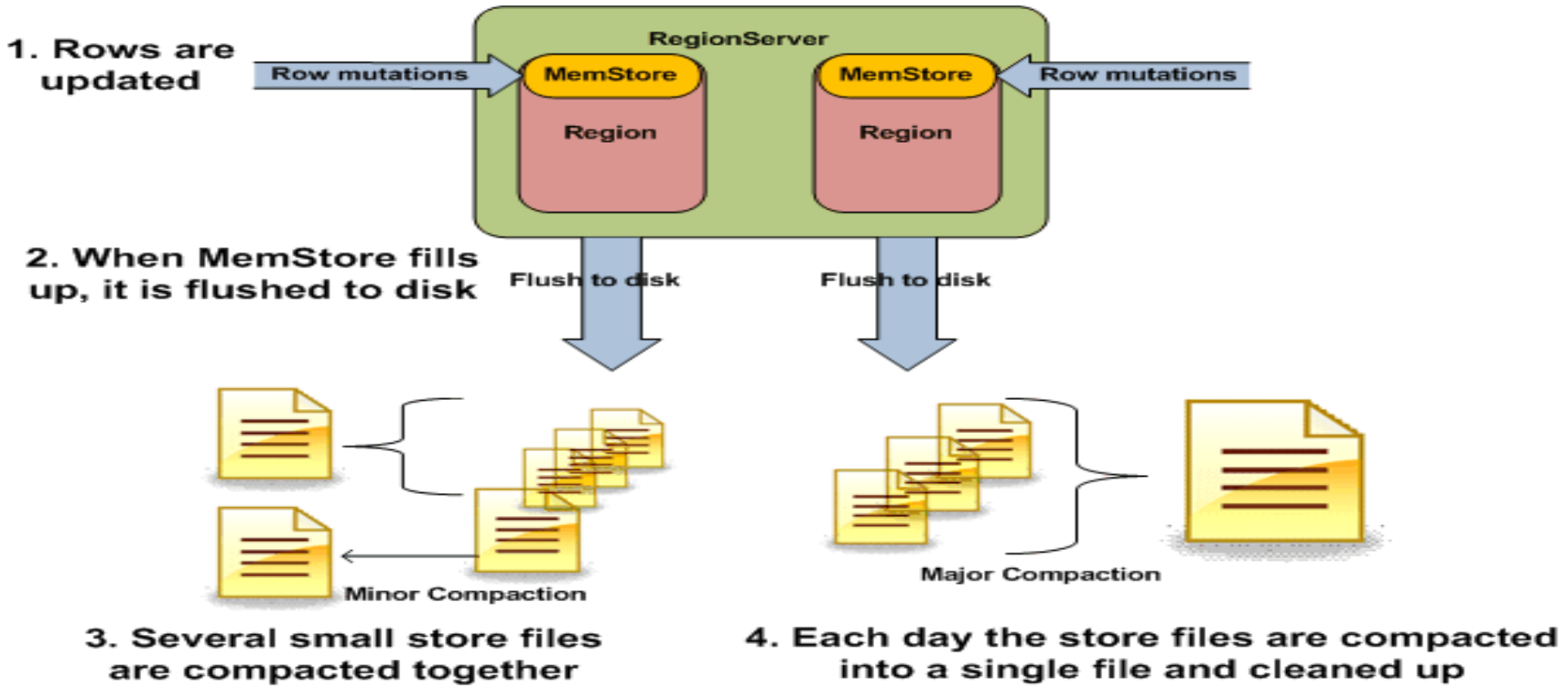
Minor Compactions

- Minor compactions combines several store files into a single file
- Usually runs after 3 store files have accumulated
- can be configured with hbase.hstore.compactionThreshold, larger number of results in fewer compactions but compaction will take longer
- Memstore cannot flush to disk during compaction, if memstore runs out of memory clients will block.

Major Compactions

- reads all the store files for a region and writes to a single store
- deleted rows and expired versions are removed
- happens once daily
- can be configured with hbase.hregion.majorcompaction default is 86400000 (24 hours), set to 0 to disable automatic major compaction
- Heavy weight operation – run when load is low

Major and Minor Compactions



Cognizant

HBase Internals

Zookeeper

Stores global information about the cluster

-ROOT-

A table that lists the .META. Tables

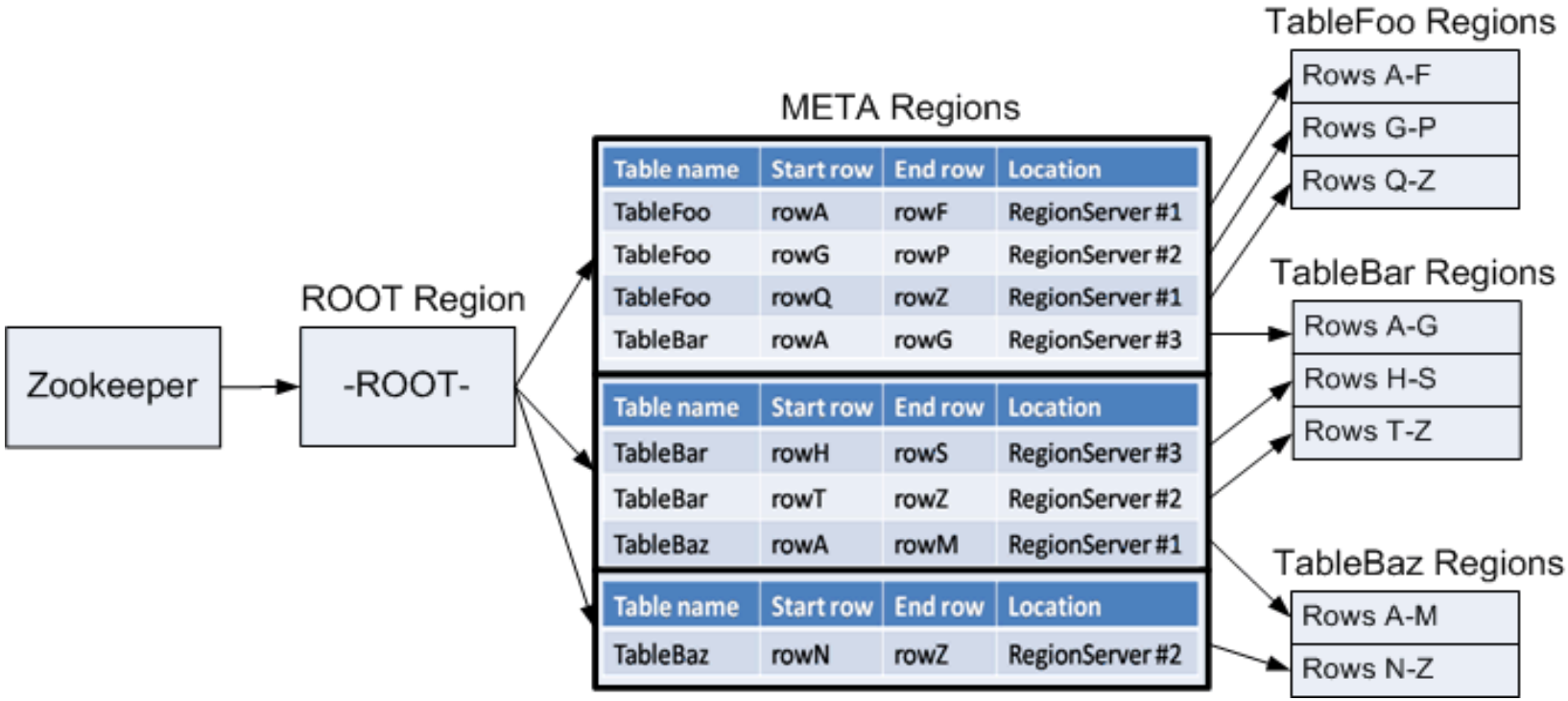
.META.

A table that lists all the regions and their locations

Reading and Writing to HBase

- HTable Client
  - Finds region servers that serve the row range of interest
  - queries zookeeper to find the location of HMaster and –ROOT-
  - queries .META table to find the region server hosting region of interest
  - Client then directly contacts region server and issues read or write request

Finding a Row

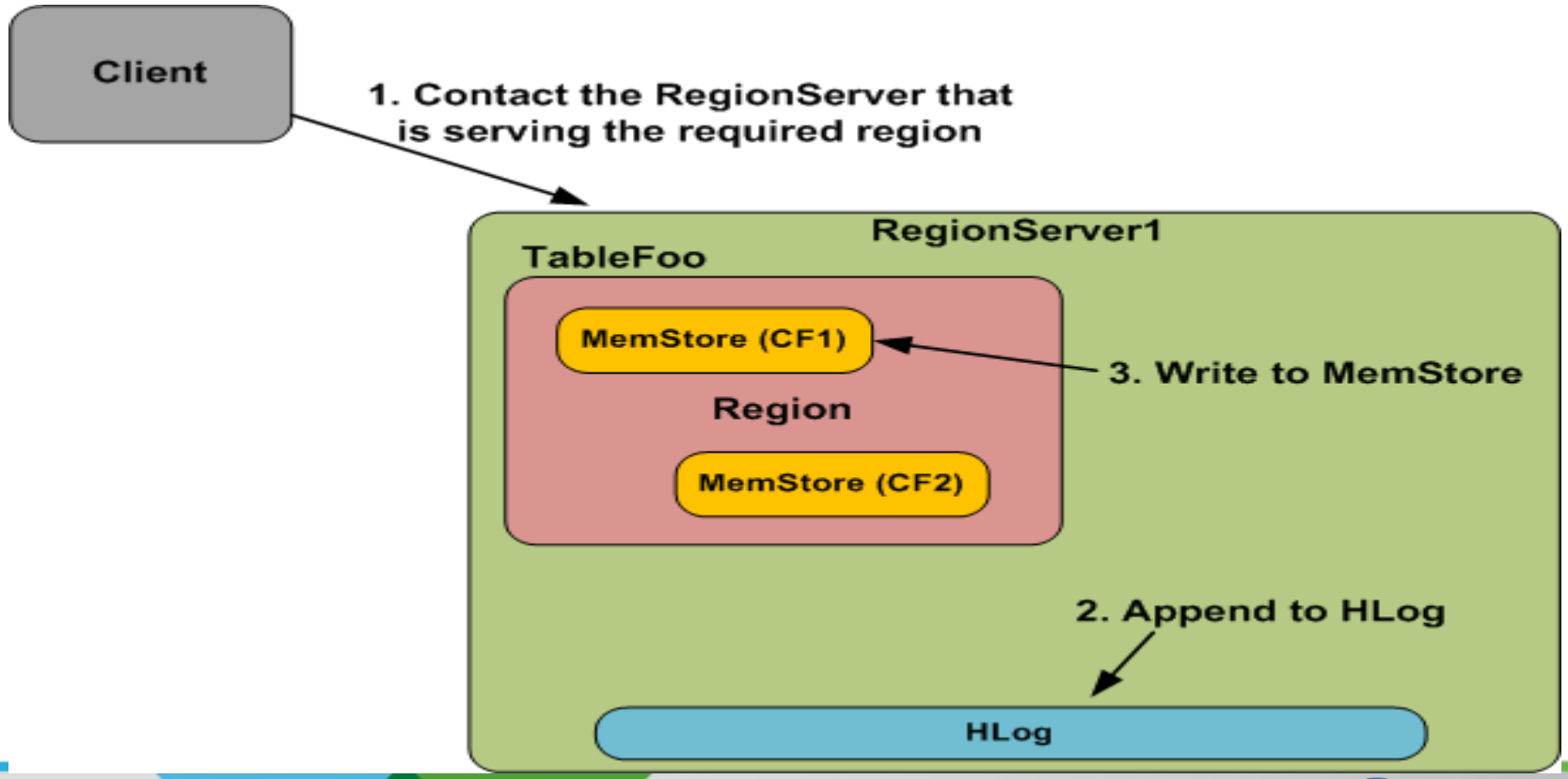


Writing Data to a Storage

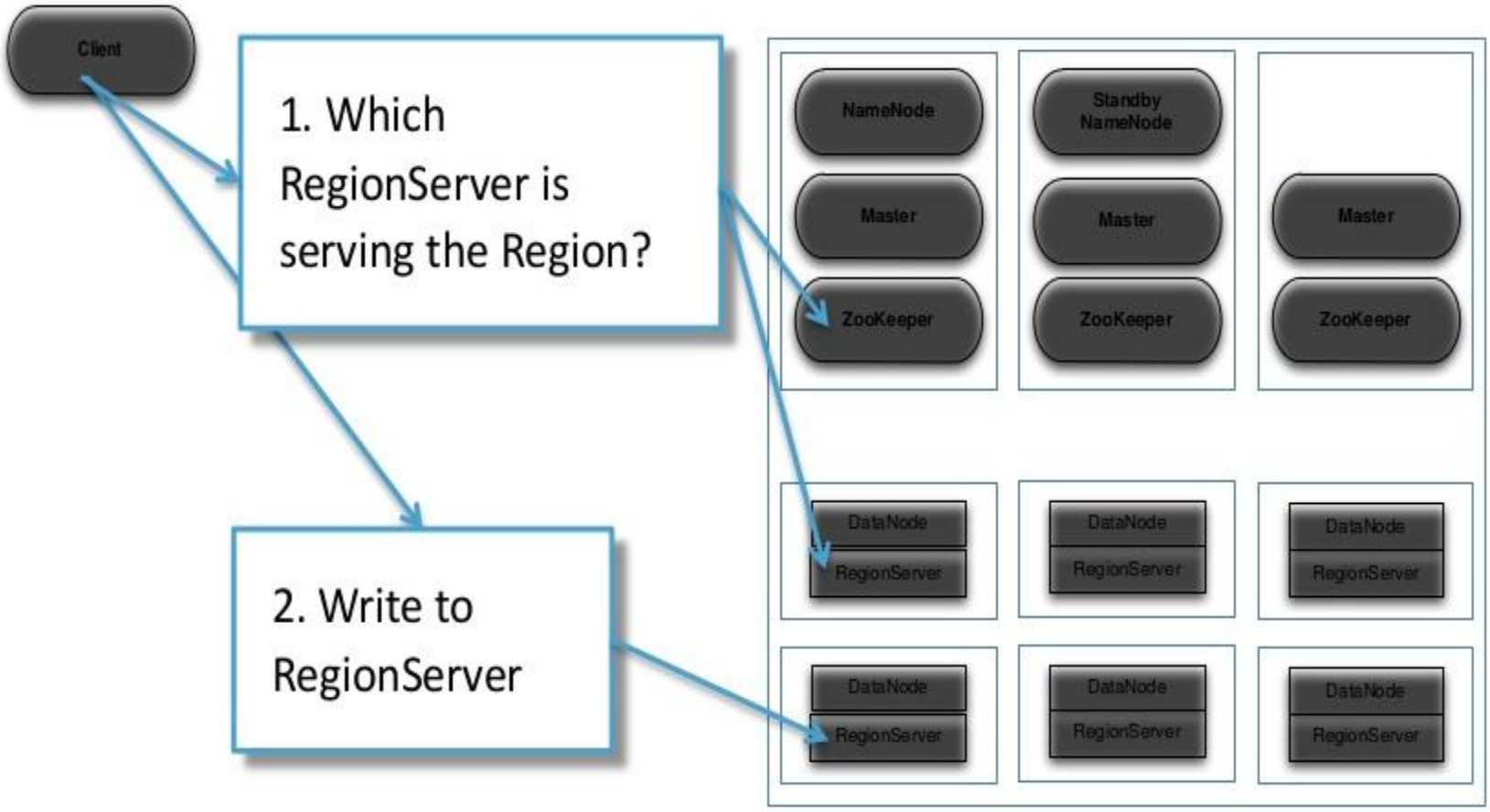
– Client issues a “put” request to region server
– details are handled to appropriate region instance
– if client flag Put.writeToWAL(‘true’) is set then the data is written to write-ahead-log (WAL) then the memstore

1. if Memstore is full then request a flush to disk
2. Flush is served by a separate region server thread
3. Flushed data is written the WAL in HDFS
4. Sequence number is saved to keep track of persisted data

Modifying a row in a Table

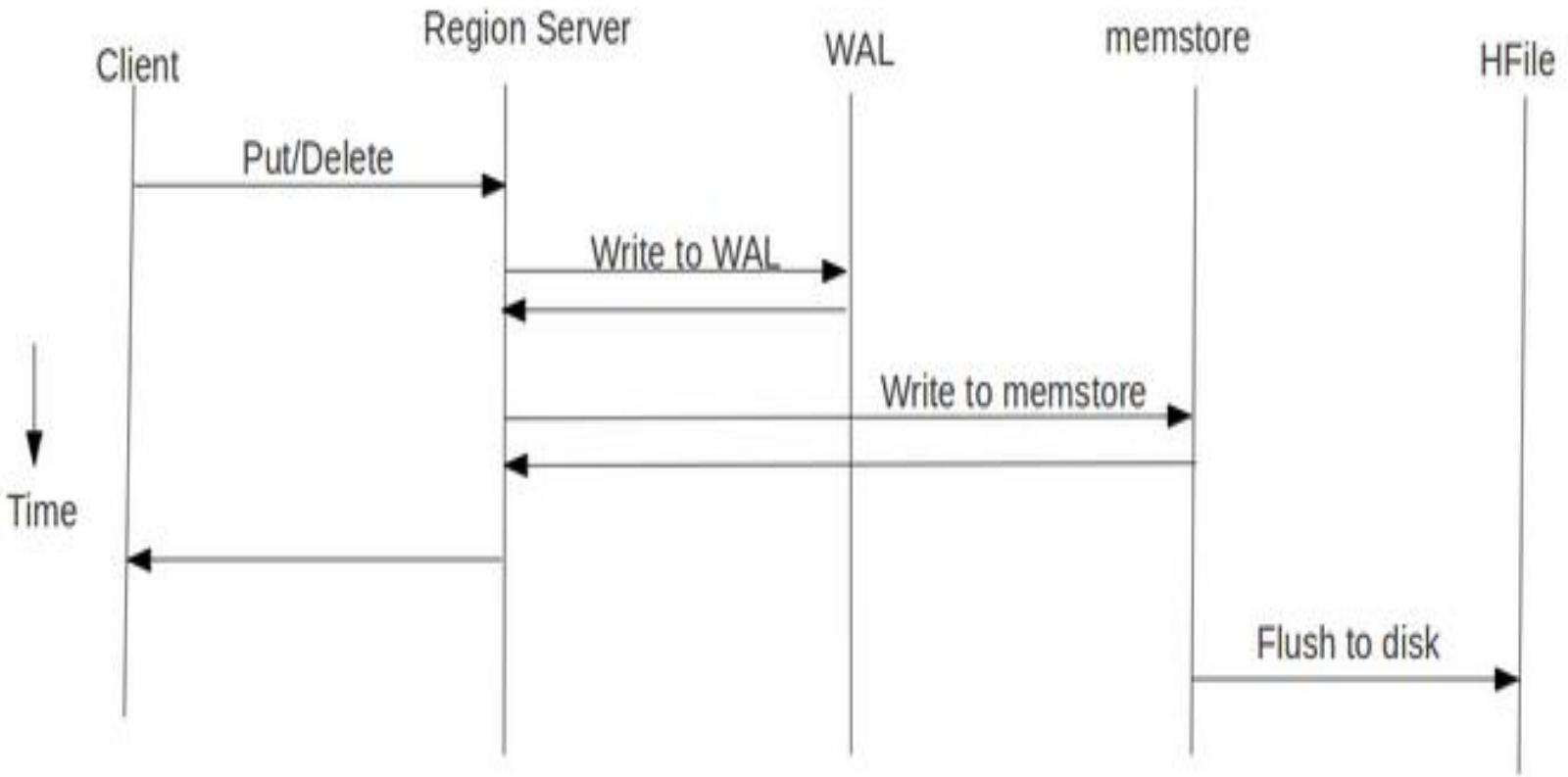


Write Path

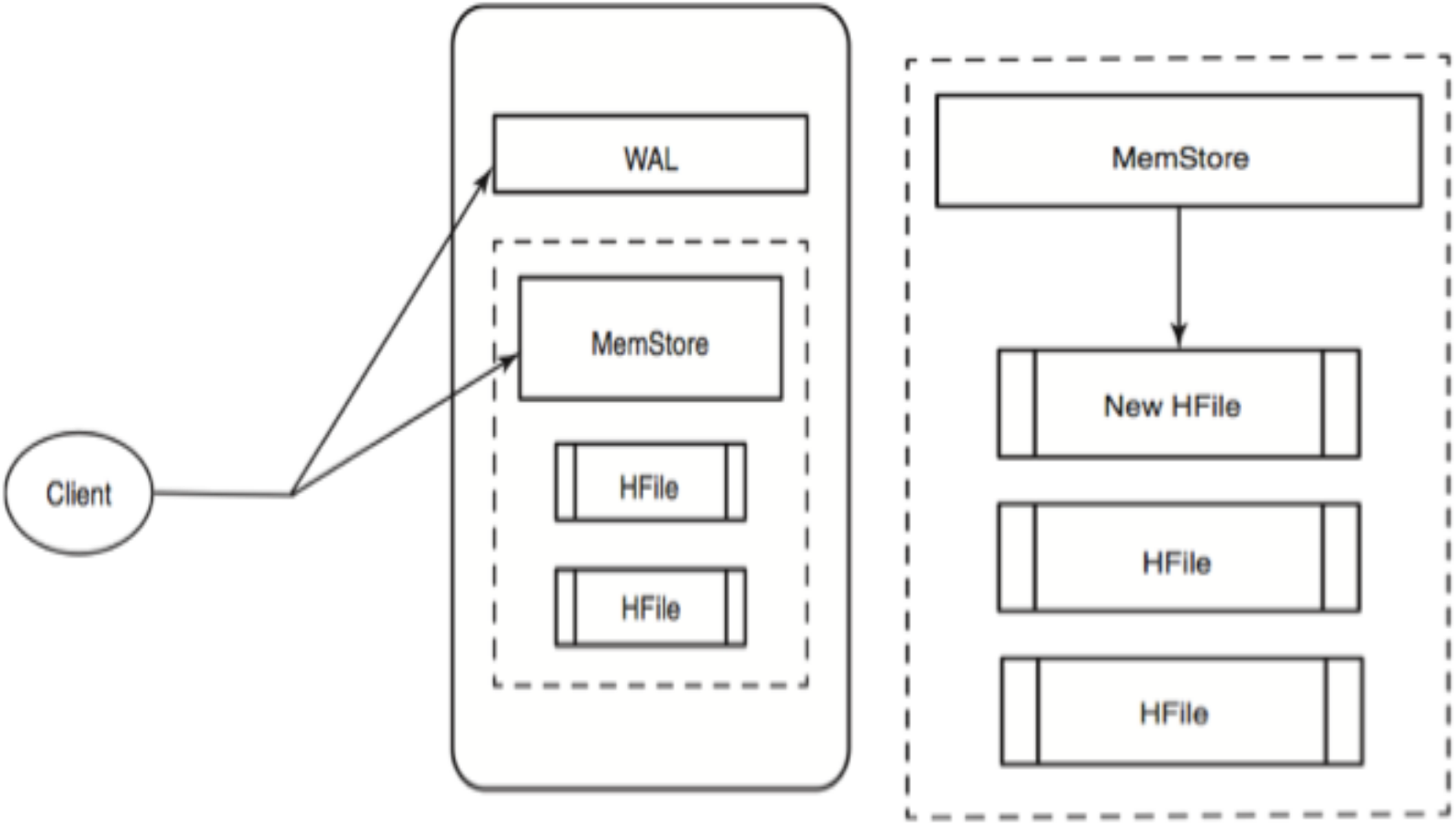


Cognizant

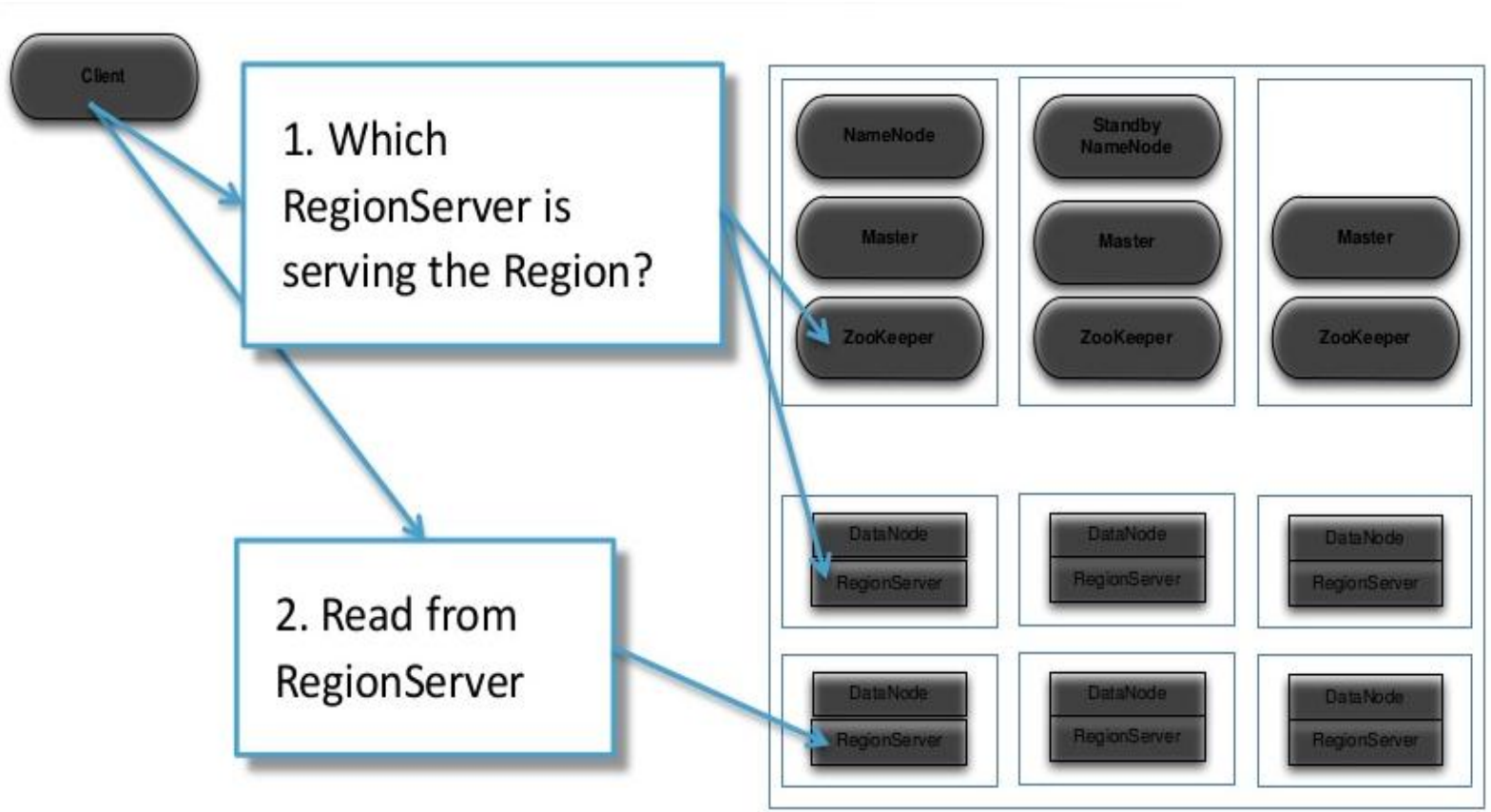
Write Path



Write Path

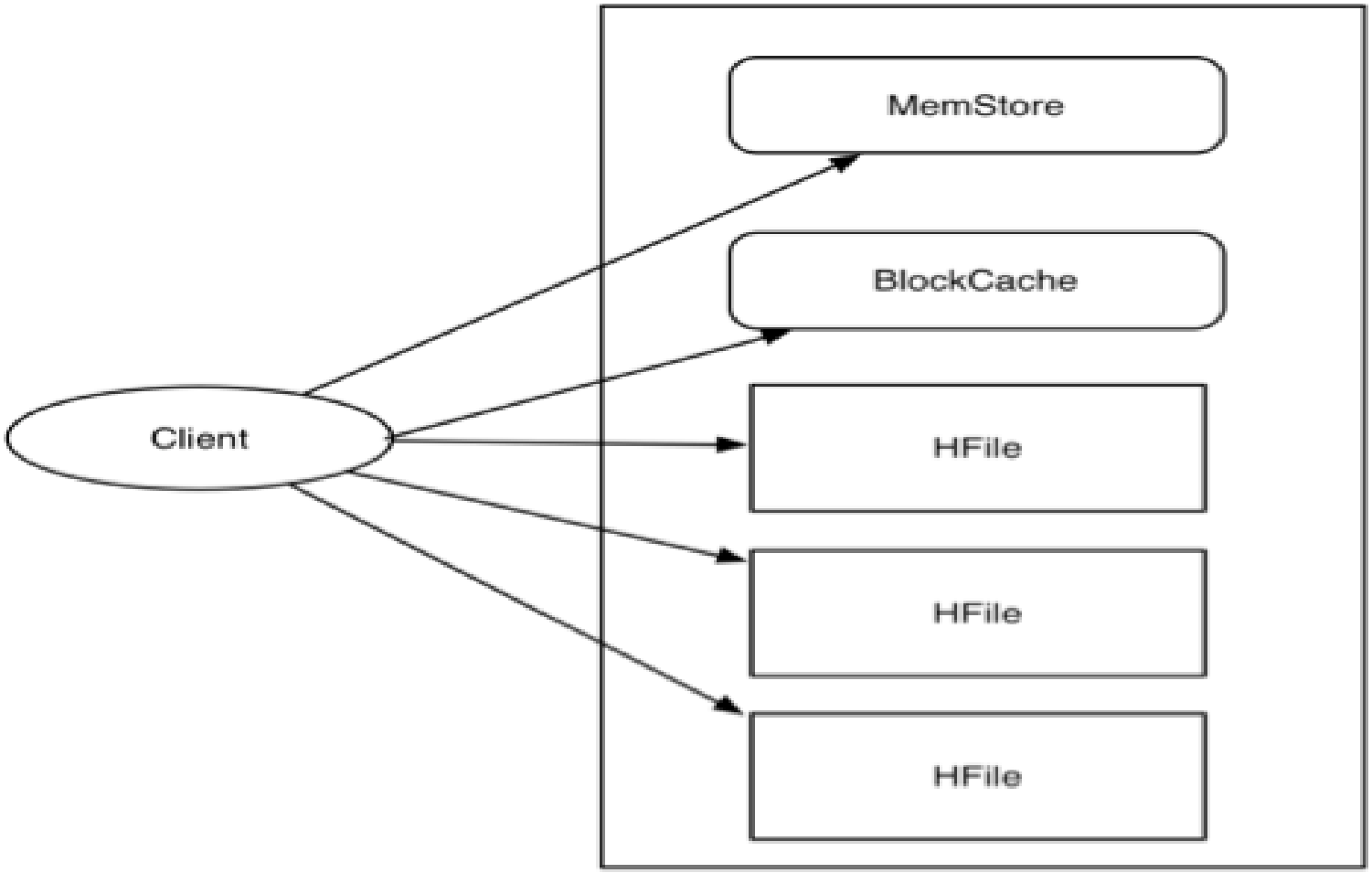


Read Path



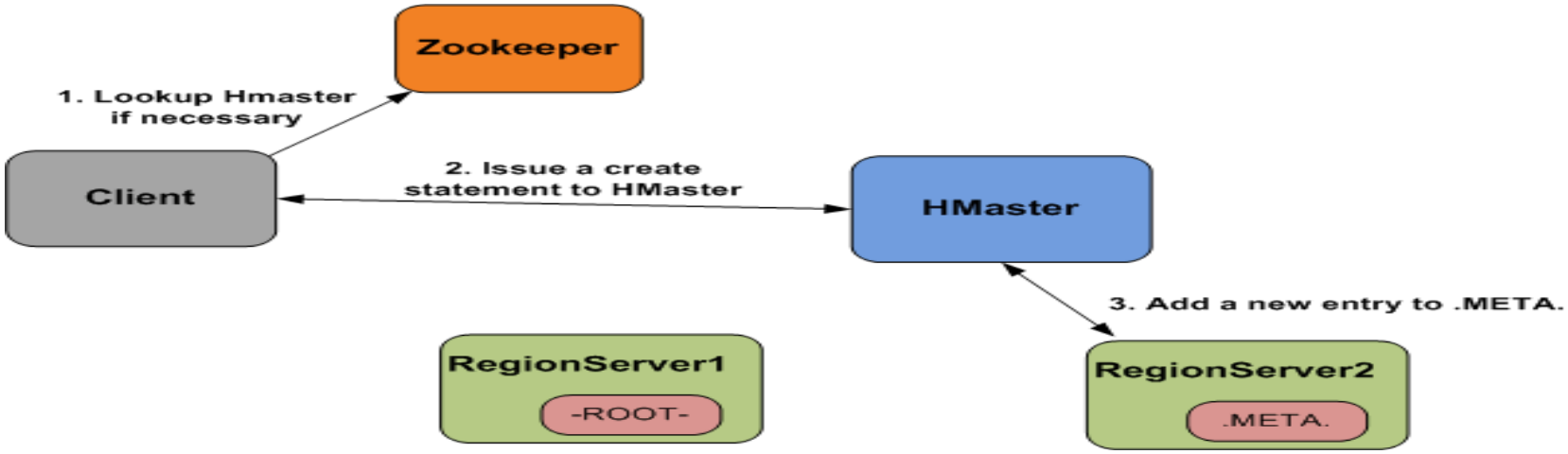
Cognizant

Read Path



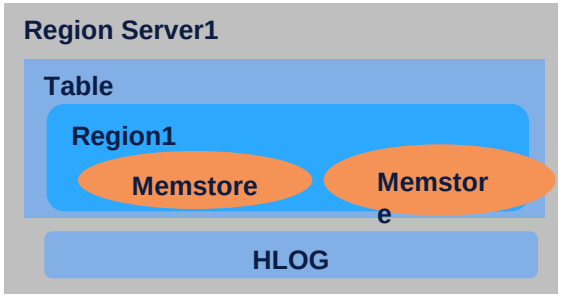
Create a Table in HBase

- Client issues create command
  1. looks up the location of master in zookeeper if necessary
  2. create command is sent to master
  3. master connects to region server serving .META and adds a new entry representing first region of new table, assigns region to region server
  4. master creates a new directory in HDFS for the region



WAL and Crash recovery

- HBase has 2 files say HLog and HFile
  - HLog : used for Write ahead log (WAL)
    standard Hadoop sequence file
    store HLogKeys as well as actual data
    HLogKeys are used to replay not yet persisted data after server crash.
  - HFile : used for actual data storage (store files)
    default block size is 64KB



WAL

- Region server updates (puts, deletes)
  - added to WAL before written to memstore
  - ensures durable writes
- HLog is WAL implementation and 1 WAL instance per region server
- 1 WAL per region results in writing too many files at same time

Region Server Crash

- Memstore is in memory and its data is lost
  - Recent changes would still be in memstore
  - WAL is used to reapply the recent changes
- Region store files (HFiles) are stored in HDFS, replicates storage blocks automatically by default
- HMaster detects the region server failure
  - begins the process of log splitting, reassigns regions to another region server and region server applies the WAL

Log Splitting

- HBase 0.90.x master splits WAL into a per region log
- HBase 0.92.x performs distributed log splitting
- when a region is opened it first checks for recovered log files, opens the per region log files and reads the edits

Bloom Filters and Block Cache

- Brute Force Approach
  - Region server reads the memstore and all store files
- HBase supports Bloom Filters
  - Eliminates the need to read every store file
  - uses a probability structure to possibly skip one or more store files
  - allows region server to skip files that do not contain row
- Bloom filter can report that a row is definitely not in the file
  - cannot guarantee that a row is in the file
  - effectively returns 'no' or 'maybe'

Bloom Filters

- Use Cases – Access patterns with lots of misses during reads
  speed up reads by cutting down internal lookups
  update all of rows regularly
  update in batches, each row is written into only a few files at a time

- Bloom filters are generated when store files are persisted
  - stored in the metadata of each store file
  - never needs to be updated as store files are immutable and adds minimal overhead to storage
- Access
  - when store file is opened the bloom filter is loaded into memory
  - used to determine if a given key is in that store file
  - can be scoped on a row key or column key level, column key needs more space as it stores many row keys
- Applying Bloom Filters
  - enabled on a per-column family basis
    - NONE : it is by default
    - ROW : Hash of the row added to bloom filter on each insert
    - ROWCOL : Hash of the row + column family + column qualifier

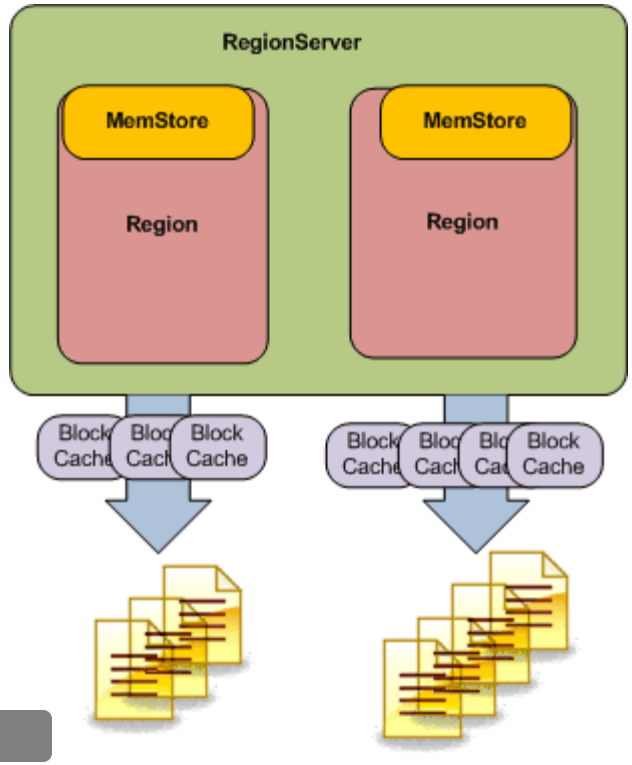
HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL)

Block Cache

- Block read from the storage files are cached internally in configurable caches
- Each store file has a block-level cache to avoid reading data from disk
- Block size is default 64K
- Memstore is write cache, Block cache is read cache
- Calculating Available cache
  - default block cache is 25% of available heap
  - default acceptable loading factor is 85%
- To calculate available cache

Number of region servers * heap size * hfile.block.cache.size * 0.85

In Block cache – your data, catalog tables, store file indexes, keys: rowkey+ family qualifier + timestamp, bloom filters



HBase Installation

- Download recent version of hbase-X.XX.X.tar.gz from hbase.apche.org
- hbase-env.sh
  - export JAVA_HOME=/usr/jdk1.6.0_32/
  - export HBASE_HEAPSIZE=1000
  - export HBASE_MANAGES_ZK=true
- hbase-site.xml
  - hbase.rootdir
  - hbase.zookeeper.property.datadir
  - fs.default.name
  - hbase.zookeeper.quorum
- regionservers
- bin/start-hbase.sh

HBase Default Ports

- **master**
  - RPC - 60000
  - UI – 60010
- **regionservers**
  - RPC - 60020
  - UI - 60030

For More Shell Commands

- **Refer in the following link:**

http://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/

http://bigdatariding.blogspot.in/2013/12/hbase-shell-commands.html

Questions

?

Thank You