Apache HBase: the Hadoop Database

Yuanru Qian, Andrew Sharp, Jiuling Wang

Agenda

- Motivation
- Data Model
- The HBase Distributed System
- Data Operations
- Access APIs
- Architecture

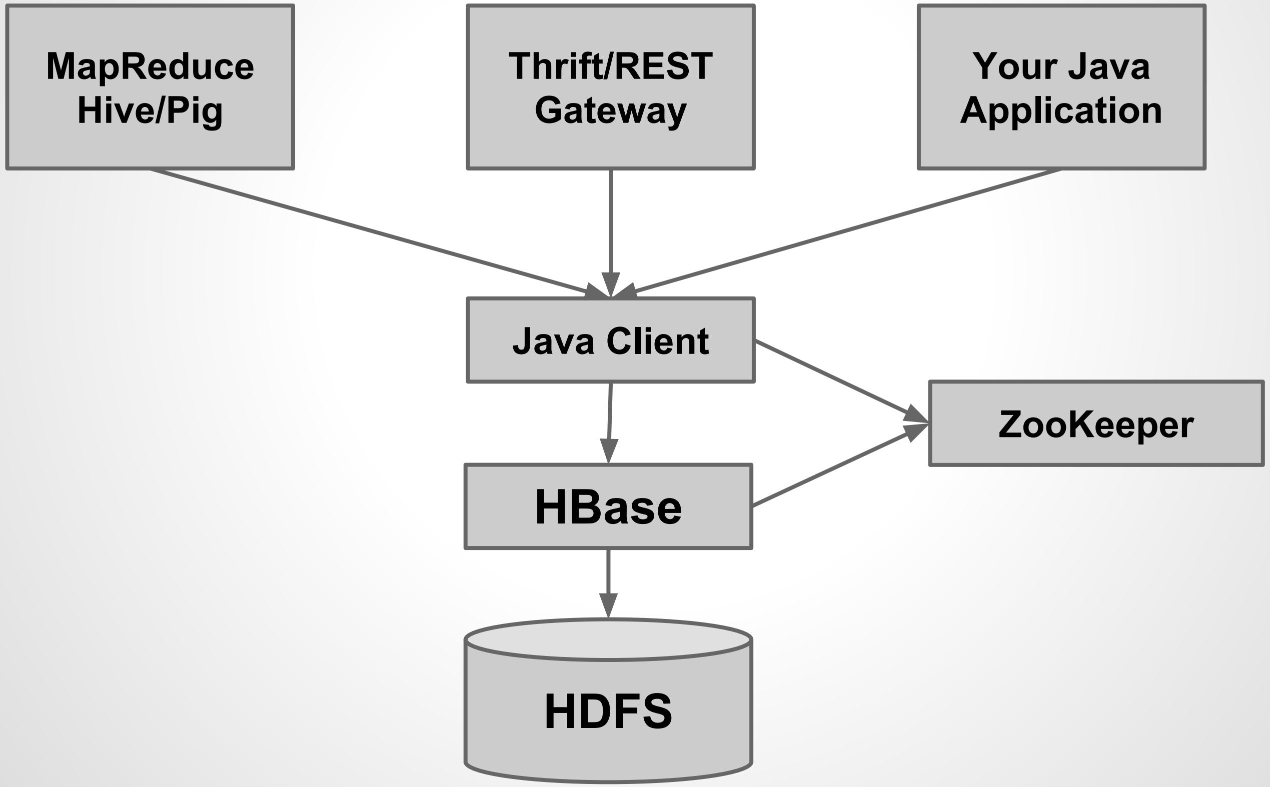
Motivation

- Hadoop is a framework that supports operations on a large amount of data.
- Hadoop includes the Hadoop Distributed File System (HDFS)
- HDFS does a good job of storing large amounts of data, but lacks quick random read/write capability.
- That’s where Apache HBase comes in.

Introduction

- HBase is an open source, sparse, consistent distributed, sorted map modeled after Google’s BigTable.
- Began as a project by Powerset to process massive amounts of data for natural language processing.
- Developed as part of Apache’s Hadoop project and runs on top of Hadoop Distributed File System.

Big Picture


MapReduce Hive/Pig
Thrift/REST Gateway
Your Java Application
Java Client
ZooKeeper
HBase
HDFS

An Example Operation

The Job:

A MapReduce job needs to operate on a series of webpages matching *.cnn.com

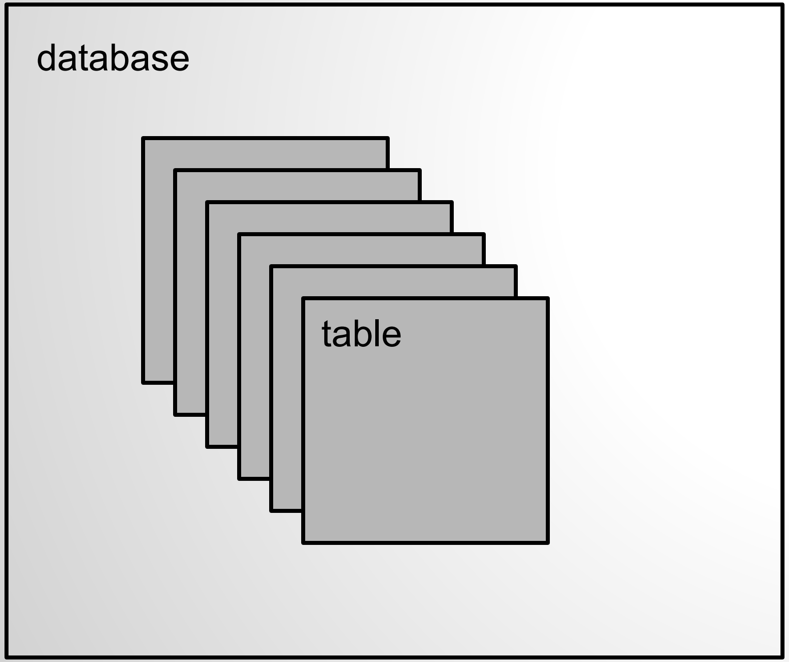
The Table:

row key	column 1	column 2
“com.cnn.world”	13	4.5
“com.cnn.tech”	46	7.8
“com.cnn.money”	44	1.2

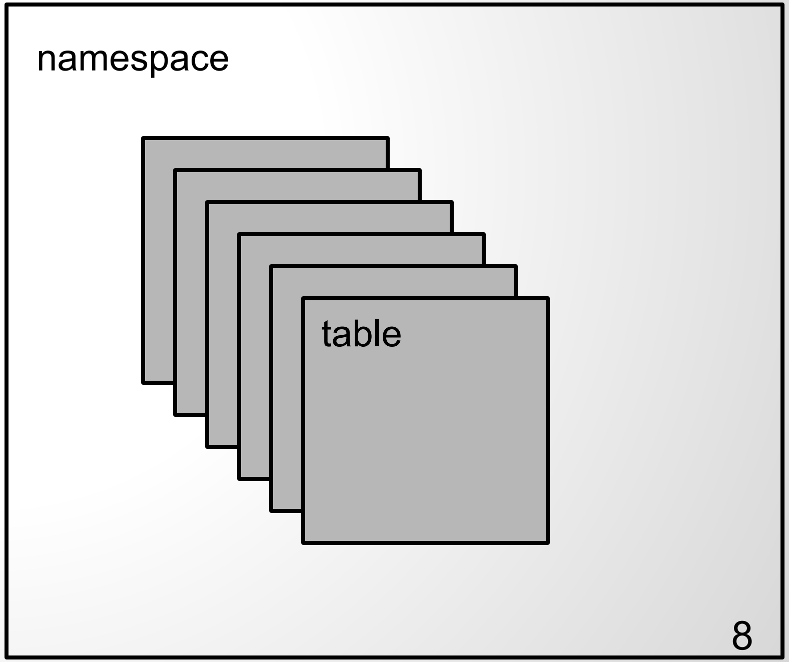
The HBase Data Model

Data Model, Groups of Tables

RDBMS



Apache HBase



Data Model, Single Table

RDBMS

table	col1	col2	col3	col4
row1				
row2				
row3				

Data Model, Single Table

columns are grouped into Column Families

Apache HBase

table	fam1		fam2	
	fam1:col1	fam1:col2	fam2:col1	fam2:col2
row1				
row2				
row3				

Sparse example

Row Key	fam1:contents	fam1:anchor
“com.cnn.www”	contents:html = “<html>...”	
	contents:html = “<html>...”	
“com.bbc.www”		anchor:cnnsi.com = "BBC"
		anchor:cnnsi.com = "BBC"

Data is *physically* stored by **Column Family**

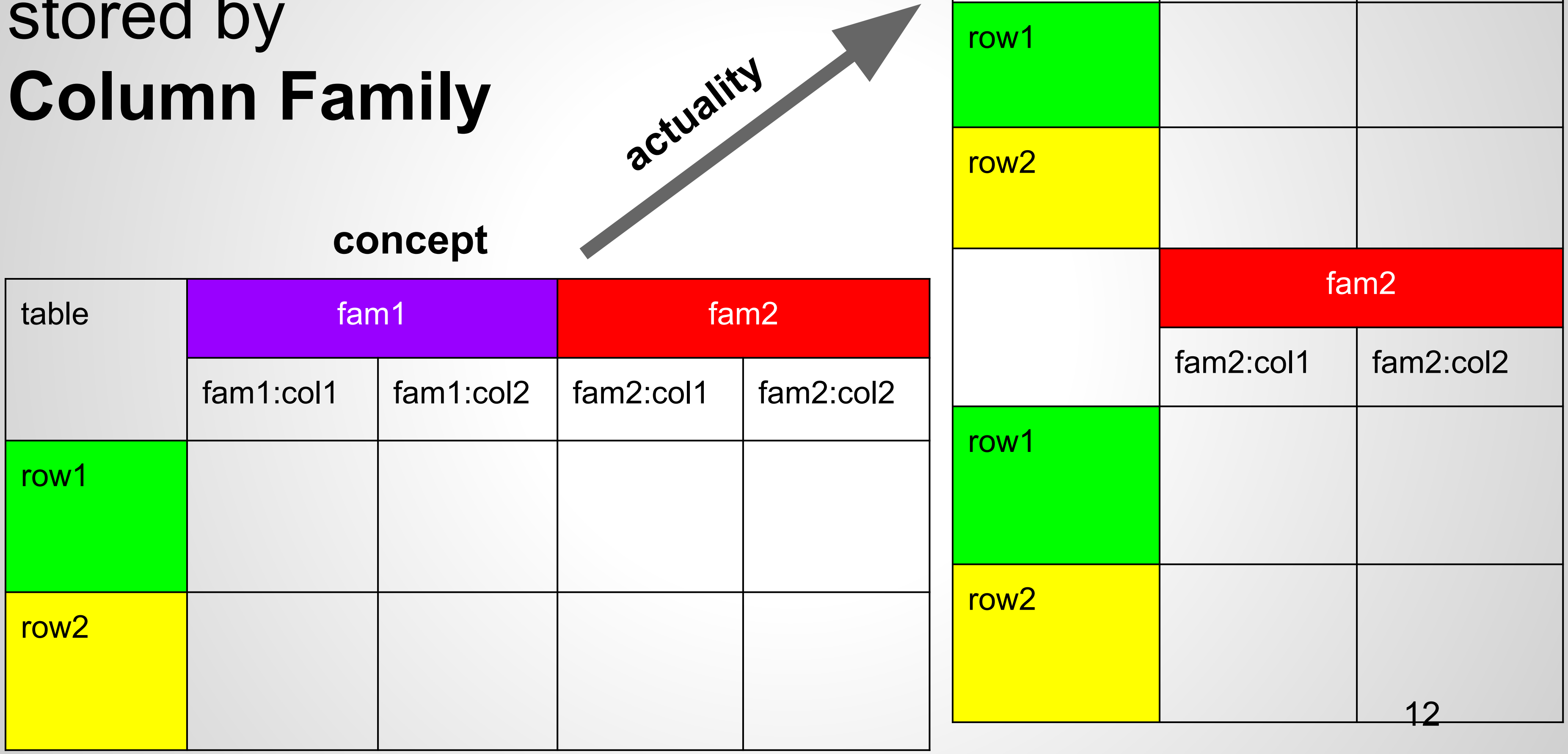


table	fam1		fam2	
	fam1:col1	fam1:col2	fam2:col1	fam2:col2
row1				
row2				

table	fam1	
	fam1:col1	fam1:col2
row1		
row2		

	fam2	
	fam2:col1	fam2:col2
row1		
row2		

Column Families and *Sharding*

Shard A

table	fam1	
	fam1:col1	fam1:col2
row1		
row2		
	fam2	
	fam2:col1	fam2:col2
row1		
row2		

Shard B

table	fam1	
	fam1:col1	fam1:col2
row3		
row4		
	fam2	
	fam2:col1	fam2:col2
row3		
row4		

Data Model, Single Table

(row, column) pairs are Versioned, sometimes referred to as Time Stamps

Apache HBase

table		fam1		fam2	
		fam1:col1	fam1:col2	fam2:col1	fam2:col2
row1	v1				
	v2				
row2	v1				
	v2				

Data Model, Single Table

A (row, column, version) tuple defines a Cell.

Apache HBase

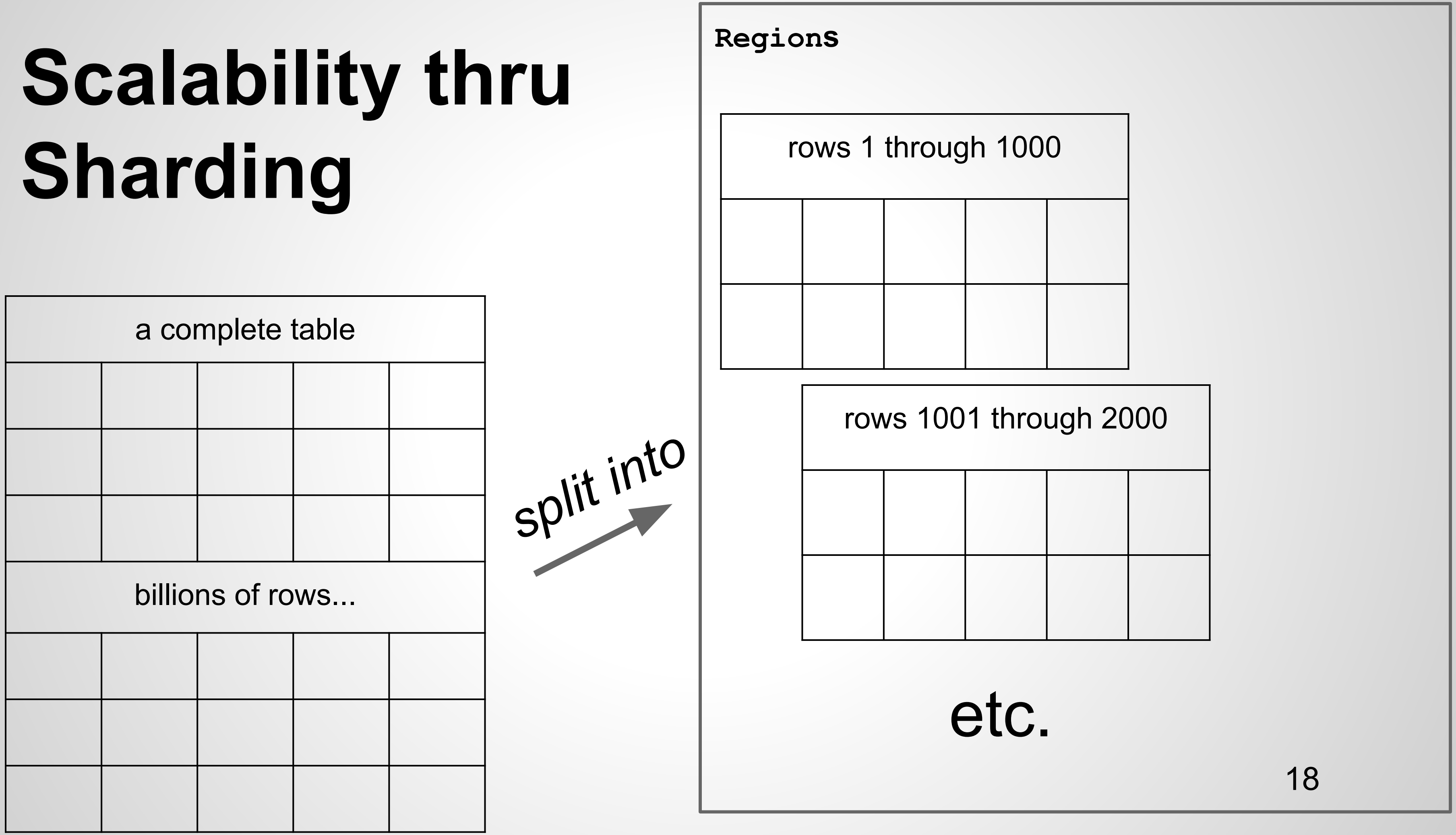
table		fam1		fam2	
		fam1:col1	fam1:col2	fam2:col1	fam2:col2
row1	v1	🟦	🟦	🟦	🟦
	v2	🟦	🟦	🟦	🟦
row2	v1	🟦	🟦	🟦	🟦
	v2	🟦	🟦	🟦	🟦

Data Model

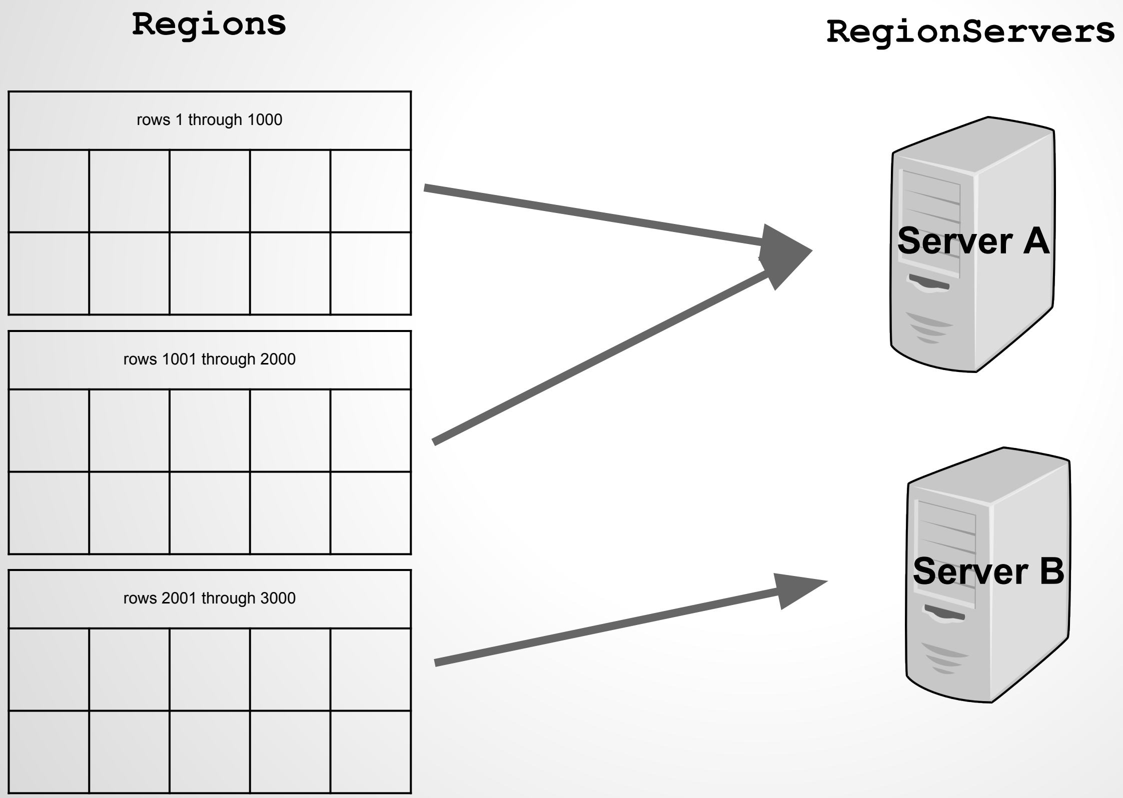
- The most basic unit is a column.
- Rows are composed of columns, and those, in turn, are grouped into column families.
- Columns are often referenced as family:qualifier.
- A number of rows, in turn, form a table, and there can be many of them.
- Each column may have multiple versions, with each distinct version contained in a separate cell.

HBase’s Distributed System

Scalability thru Sharding

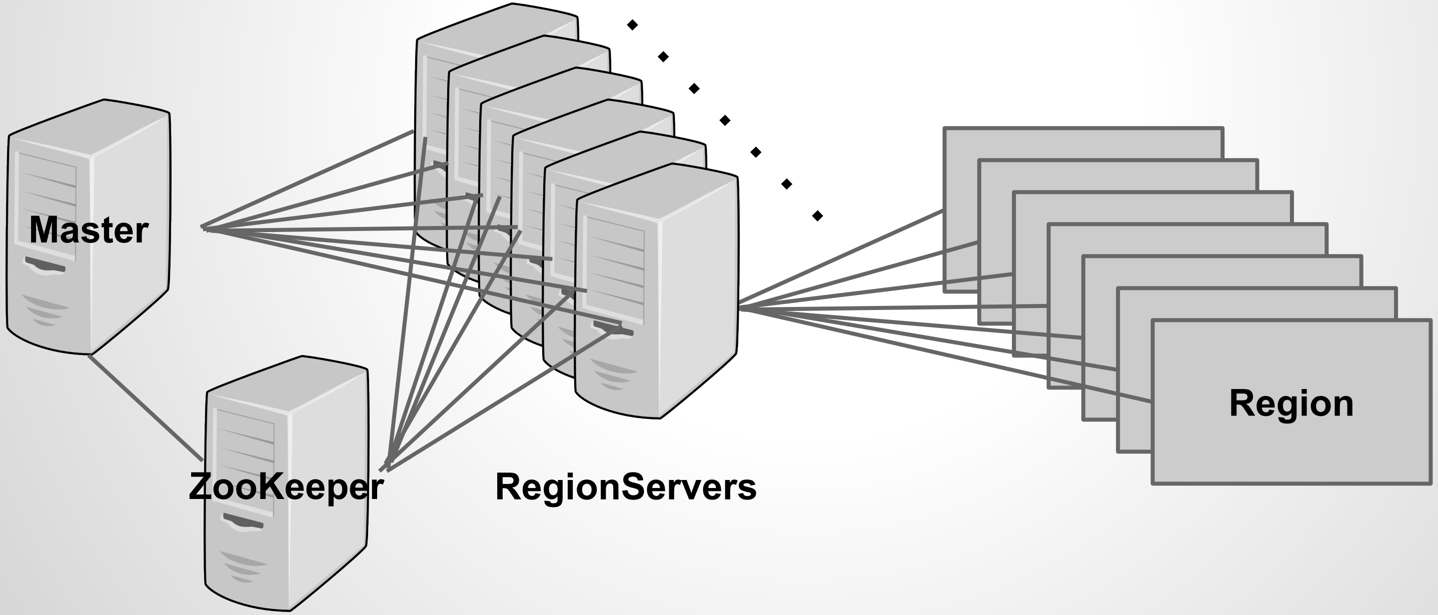


Scalability thru Sharding

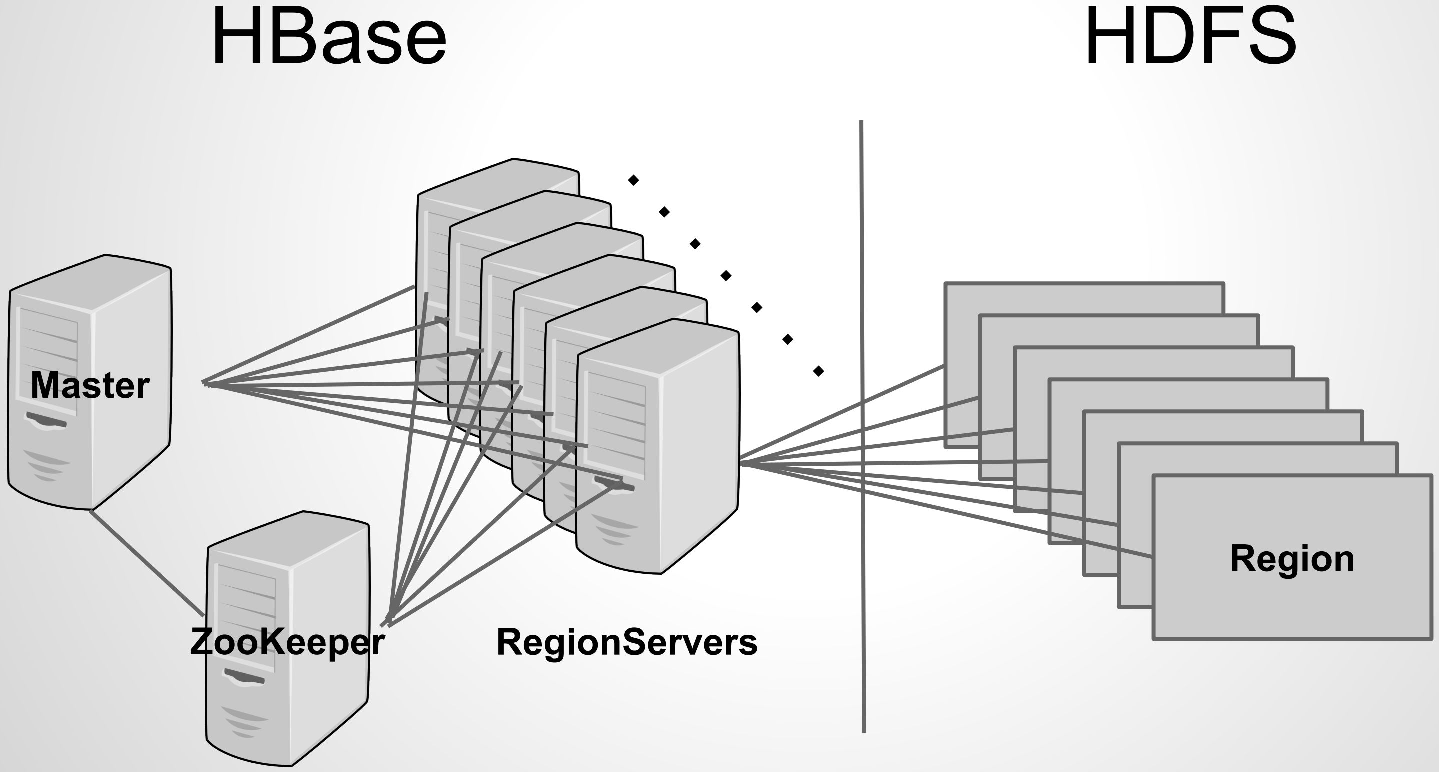


Scalability thru Division of Labor

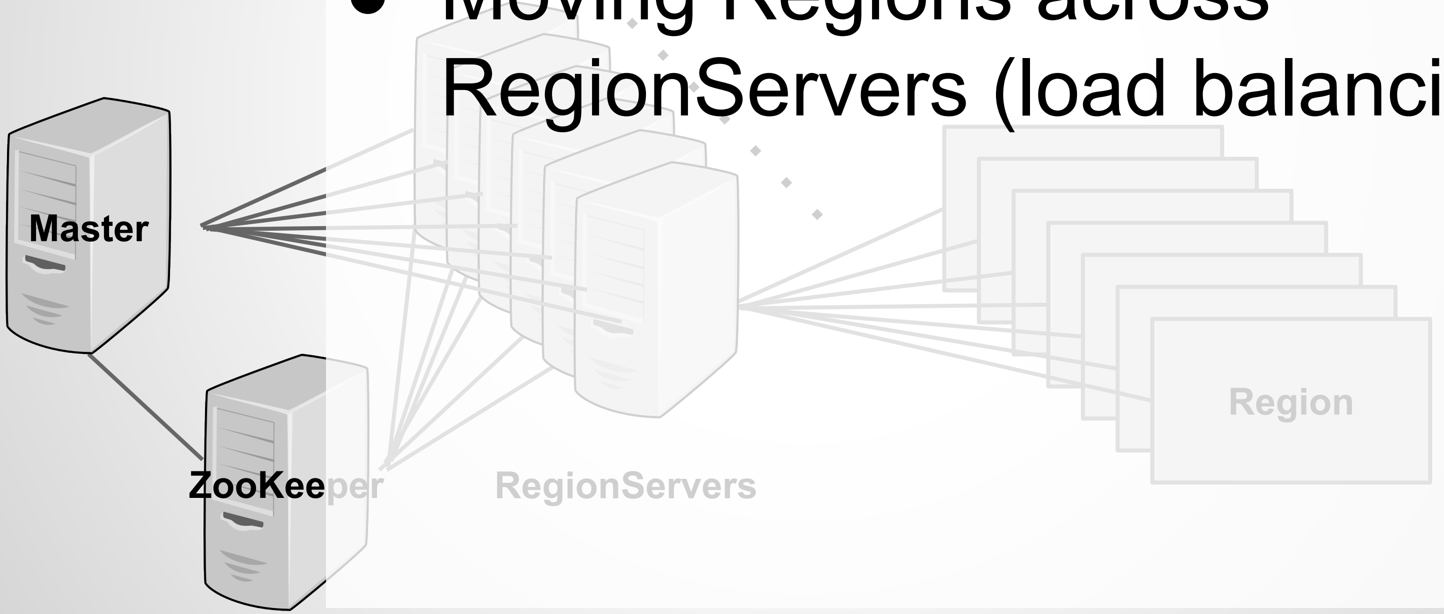
An HBase Distributed System



Scalability thru Division of Labor



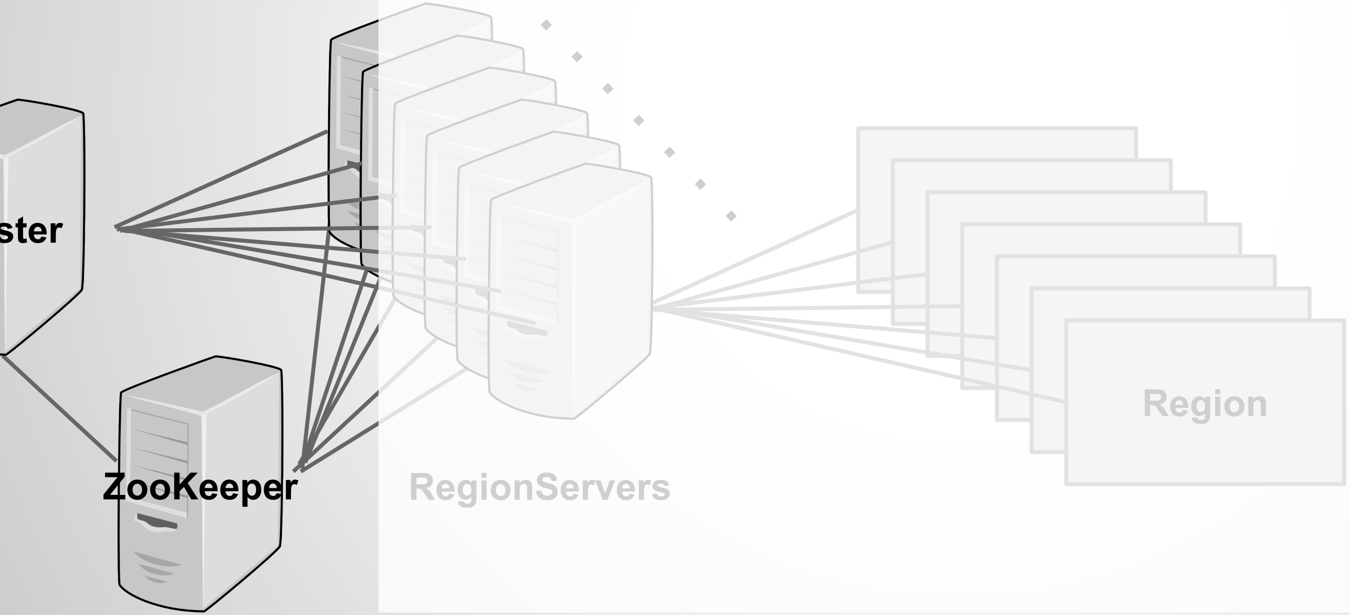
Division of Labor, Master



- Schema changes
- Moving Regions across RegionServers (load balancing)

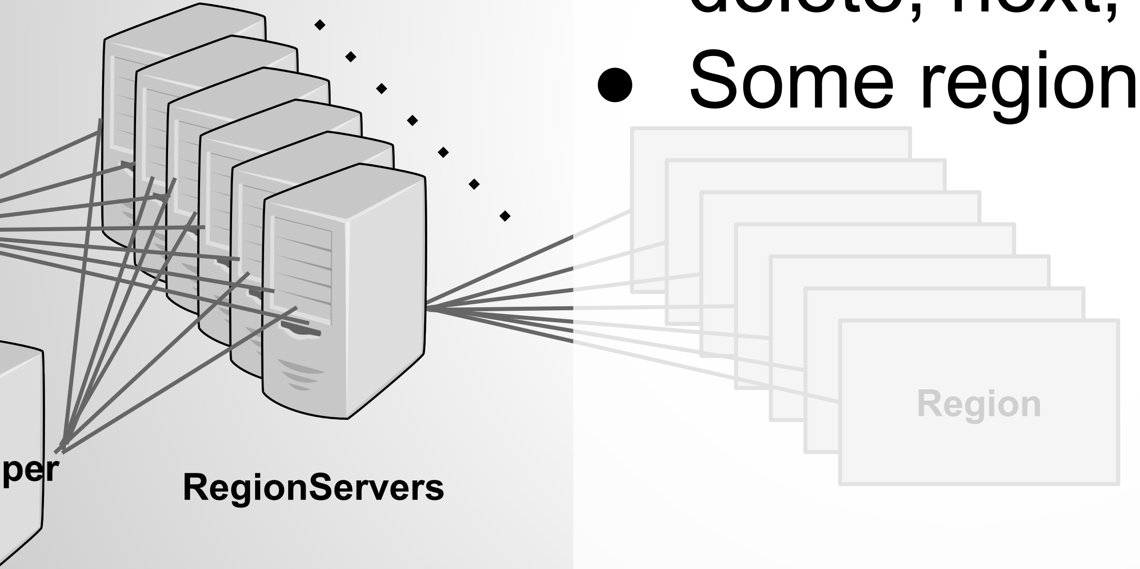
Division of Labor, ZooKeeper

- Locating Regions



Division of Labor, RegionServer

- Data operations (put, get, delete, next, etc.)
- Some region splits



The HBase Distributed System

Region

- a subset of table's rows, like a range partition
- Automatically sharded

RegionServer

- Servers data for reads and writes for a group of regions.

Master

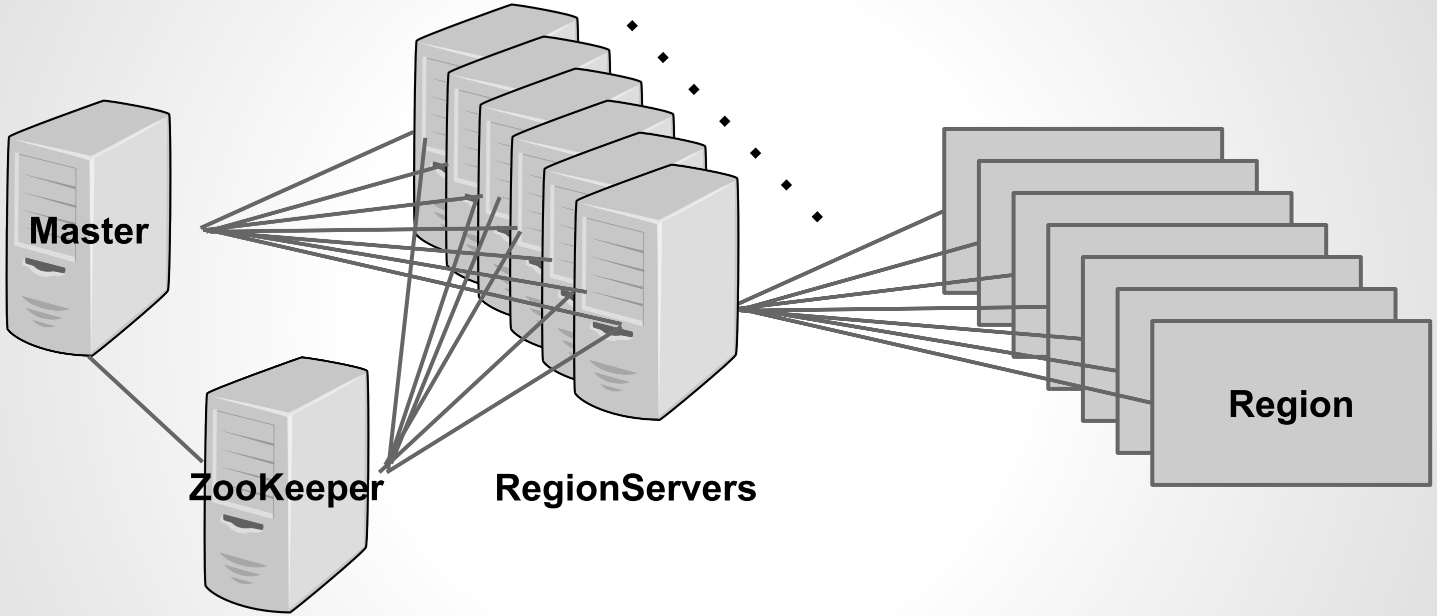
- Responsible for coordinating the RegionServers
- Assign regions, detects failures of RegionServers
- Control some admin functions

ZooKeeper

- Locate data among RegionServers

Availability thru Automatic Failover

- DataNode failures handled by HDFS(replication)
- RegionServer failures handled by Master re-assigning Regions to available RegionServers.
- HMaster failover is handled automatic by having multiple HMasters.


Master
ZooKeeper
RegionServers
Region

How to Access to HBase?

Java Client Interfaces

- **Configuration** holds details where to find the cluster and tunable settings. Roughly equivalent to JDBC connection string.
- **HConnection** represents connections to the cluster.
- **HBaseAdmin** handles DDL operations(create,list,drop,alter,etc)
- **HTable** is a handle on a single HBase table. Send “commands” to the table.(Put,Get,Scan,Delete).

Scan

```
//Return the result of columns called cf:qualifier from row 1 to row 1000.
HTable table = ...    // instantiate HTable
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf"),Bytes.toBytes("qualifier"));
scan.setStartRow( Bytes.toBytes("row1"));  // start key is inclusive
scan.setStopRow( Bytes.toBytes("row1000")); // stop key is exclusive
ResultScanner scanner = table.getScanner(scan)
try {
    for(Result result : scanner) {
    // process Result instance
    }
} finally {
    scanner.close();
}
```

Scan

```
//Return the result of column family called cf from row 1 to row 1000
HTable table = ...    // instantiate HTable
Scan scan = new Scan();
scan.addFamily(Bytes.toBytes("cf"));
scan.setStartRow( Bytes.toBytes("row1"));  // start key is inclusive
scan.setStopRow( Bytes.toBytes("row1000")); // stop key is exclusive
ResultScanner scanner = table.getScanner(scan)
try {
    for(Result result : scanner) {
    // process Result instance
    }
} finally {
    scanner.close();
}
```

Get

Return an entire row

```
HTable htable = ...    // instantiate HTable
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
```

Return column family called cf

```
HTable htable = ...    // instantiate HTable
Get get = new Get(Bytes.toBytes("row1"));
get.addFamily(Bytes.toBytes("cf"));
Result r = htable.get(get);
```

Return the column called cf:qualifier

```
HTable htable = ...    // instantiate HTable
Get get = new Get(Bytes.toBytes("row1"));
get.addColumn(Bytes.toBytes("cf"),Bytes.
toBytes("qualifier"));
Result r = htable.get(get);
```

Return column family in version2.

```
HTable htable = ...    // instantiate HTable
Get get = new Get(Bytes.toBytes("row1"));
get.addFamily(Bytes.toBytes("cf"));
get.setTimestamp(v2);
Result r = htable.get(get);
```

Delete

Delete an entire row

```
HTable htable = ...    // instantiate HTable
Delete delete = new Delete(Bytes.toBytes("row1"));
htable.delete(delete);
```

Delete the latest version of a specified column

```
HTable htable = ...    // instantiate HTable
Delete delete = new Delete(Bytes.toBytes("row1"));
delete.deleteColumn(Bytes.toBytes("cf"),Bytes.
toBytes("qualifier"));
htable.delete(delete);
```

Delete a specified version of a specified column

```
HTable htable = ...    // instantiate HTable
Delete delete = new Delete(Bytes.toBytes
("row1"));
delete.deleteColumn(Bytes.toBytes("cf"),
Bytes.toBytes("qualifier"),version);
htable.delete(delete);
```

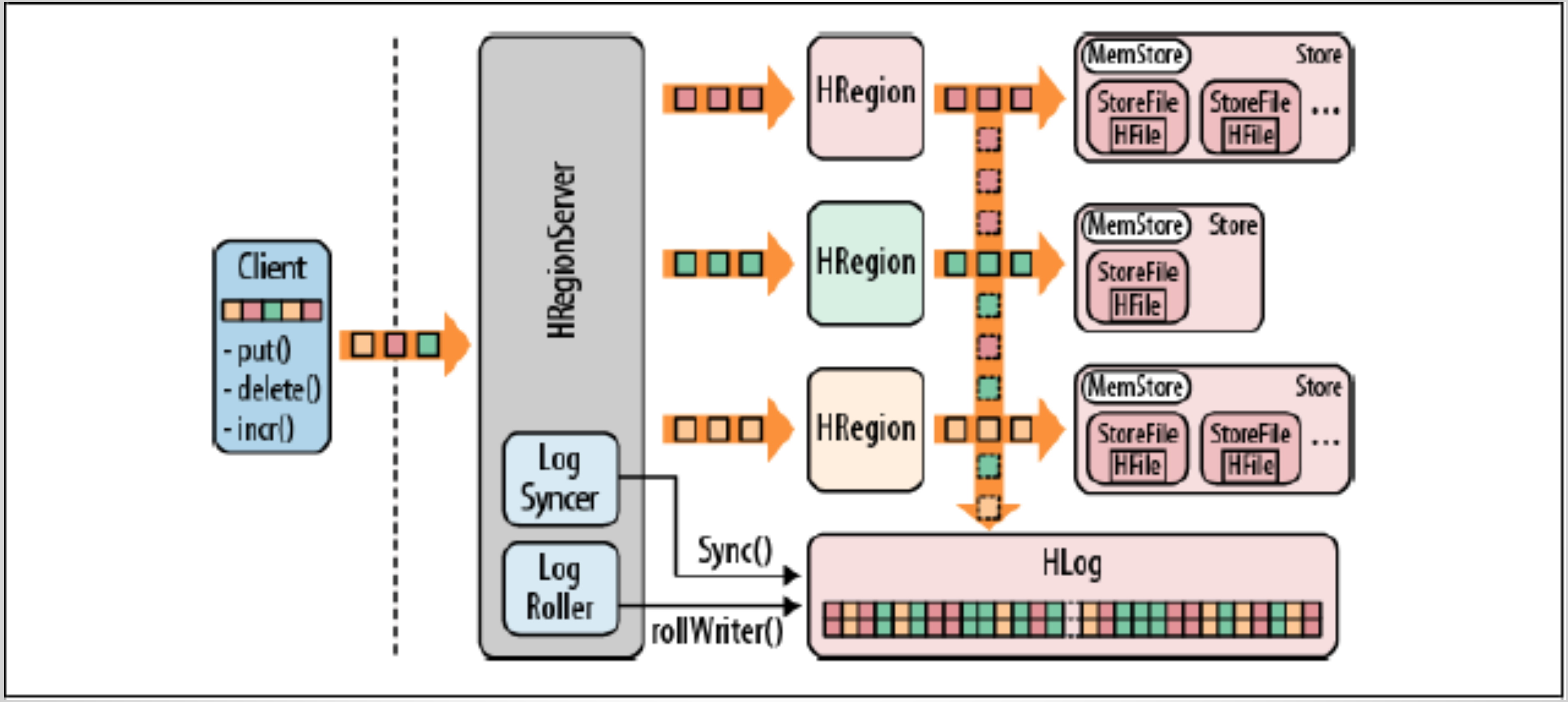
Put

Put a new version of a cell using current timestamp by default

```
HTable htable = ...    // instantiate HTable
Put put = new Put(Bytes.toBytes("row1"));
put.add(Bytes.toBytes("cf"),Bytes.toBytes
("qualifier"),Bytes.toBytes("data"));
htable.put(put);
```

Overwriting an existing value

```
HTable htable = ...    // instantiate HTable
Put put = new Put(Bytes.toBytes("row1"));
put.add(Bytes.toBytes("cf"),Bytes.toBytes
("qualifier"),timestamp,Bytes.toBytes
("data"));
htable.put(put);
```


Client
- put()
- delete()
- incr()
HRegionServer
Log Syncer
Log Roller
HRegion
HRegion
HRegion
MemStore
Store
StoreFile
HFile
StoreFile
HFile
...
MemStore
Store
StoreFile
HFile
MemStore
Store
StoreFile
HFile
StoreFile
HFile
...
Sync()
rollWriter()
HLog

Implementation Details of Put

1. The client initiates an action that modifies data.
2. Modification is wrapped into a KeyValue object instance and sent over to the HRegionServer that serves the matching regions.
3. Once the KeyValue instance arrives, they are routed to the HRegion instances that are responsible for the given rows.
4. The data is written to the Write-Ahead Log, and then put into MemStore of the actual Store that holds the record.
5. When the memstores get to a certain size, the data is persisted in the background to the filesystem.

Join?

- HBase does not support join.
    - NoSql is mostly designed for fast appends and key-based retrievals.
    - Joins are expensive and infrequent.
- What if you still need it?
    - Write a MapReduce join to make it.
    - At Map function, read two tables. The output key should be the value on the joined attribute for table1 and table2.
    - At Reduce function, “join” the tuple that contains the same key.
    - Other implementations using Hive/Pig/...

Other Clients

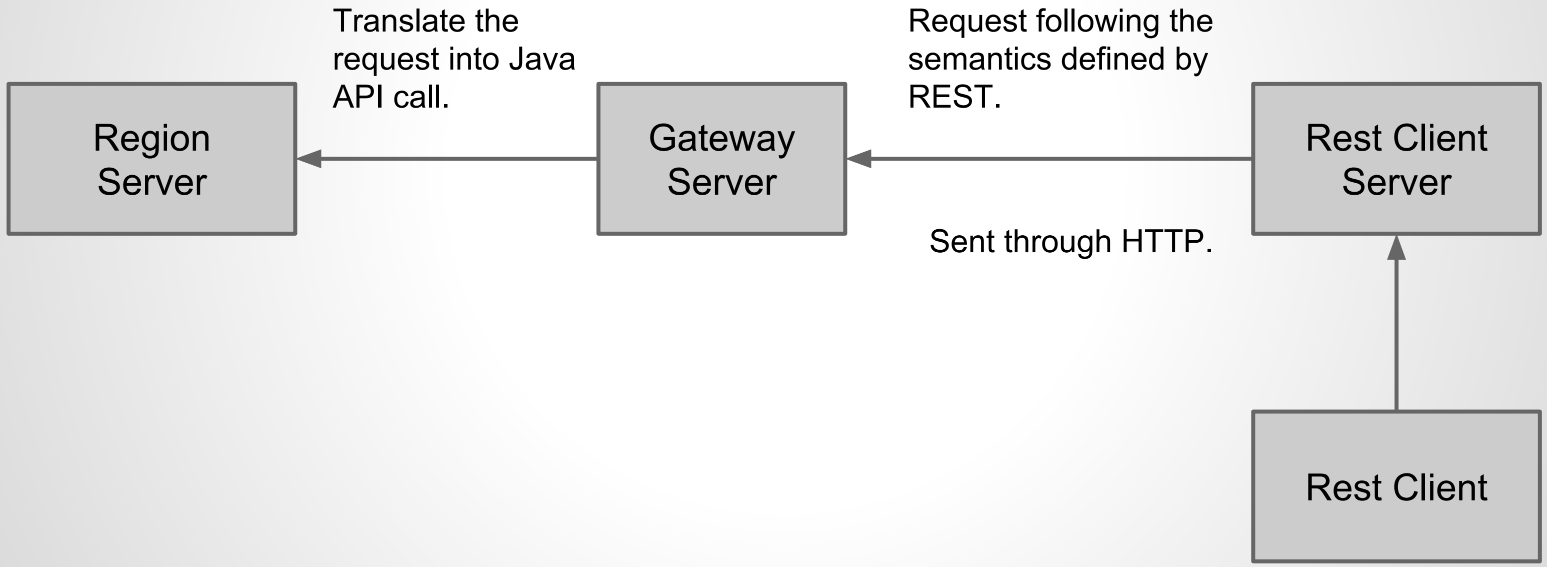
Use some sort of proxy that translate your request into an API call.

- These proxies wrap the native Java API into other protocol APIs.
- Representational State Transfer(REST)
- Protocol Buffers, Thrift, Avro

REST

- is a protocol between the gateways and the clients.
- uses HTTP verbs to perform an action, giving developers a wide choice of languages and programs to use.
- suffers from the verbosity level of the protocol. Human-readable text, be in plain or XML-based, is used to communicate between the client and server.

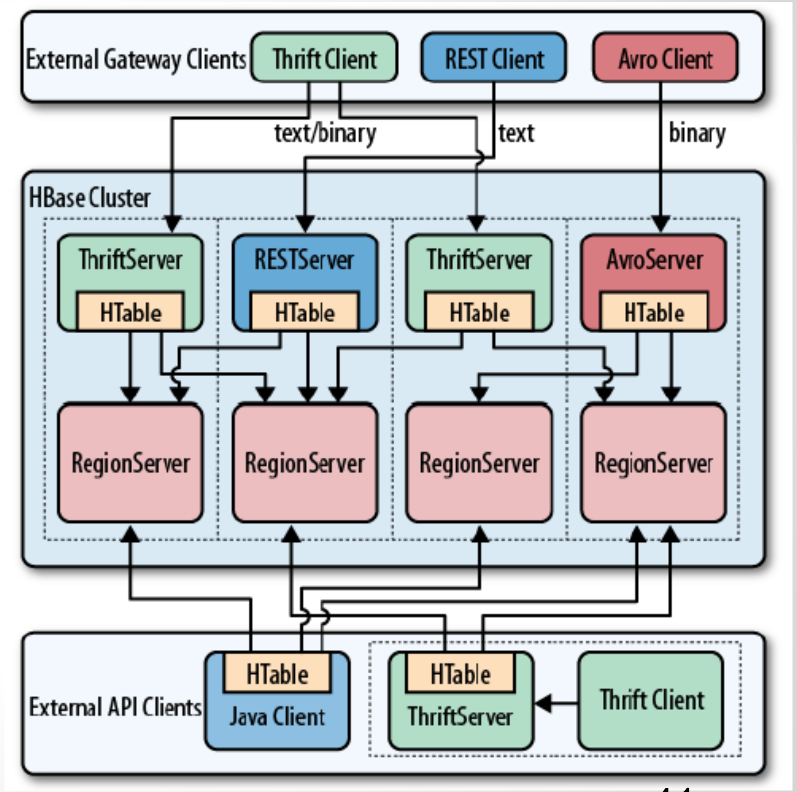
REST


Translate the request into Java API call.
Request following the semantics defined by REST.
Region Server
Gateway Server
Rest Client Server
Sent through HTTP.
Rest Client

Improvements

Companies with large server farms, extensive bandwidth usage, and many disjoint services felt the need to reduce the overhead and implemented their own Remote Procedure Call(RPC) layers.

- Google Protocol Buffers
- Facebook Thrift
- Apache Avro

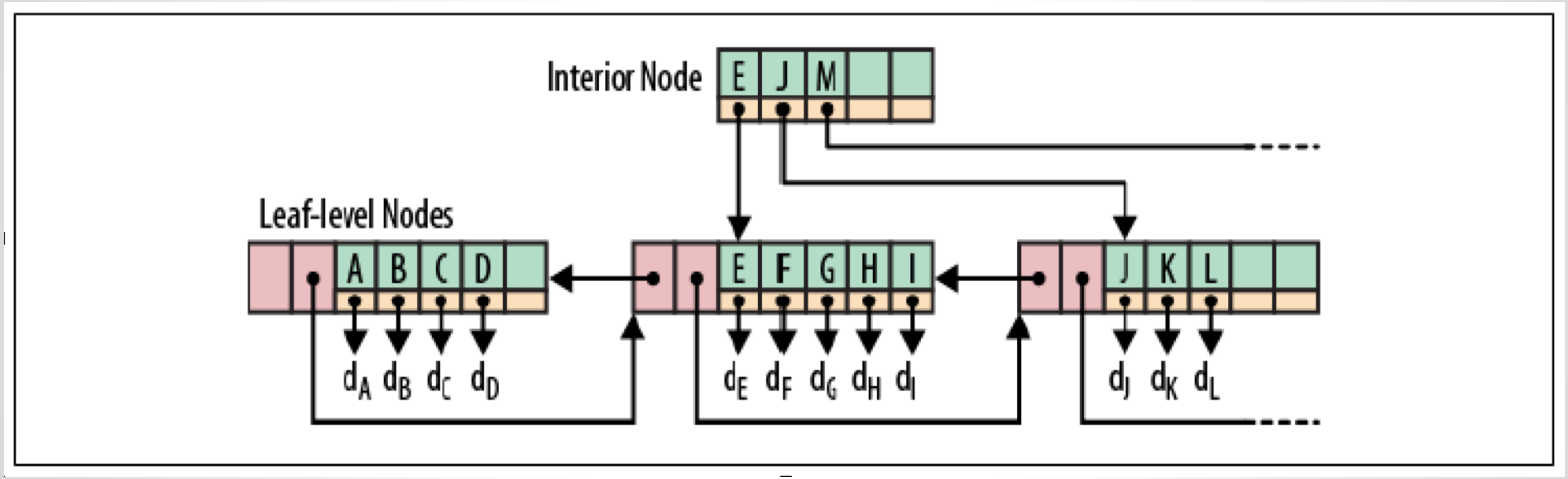


Architecture

storage structures:

- B+ Trees (typical RDBMS storage)
- Log-Structured Merge-Trees(HBase)

B+ Trees



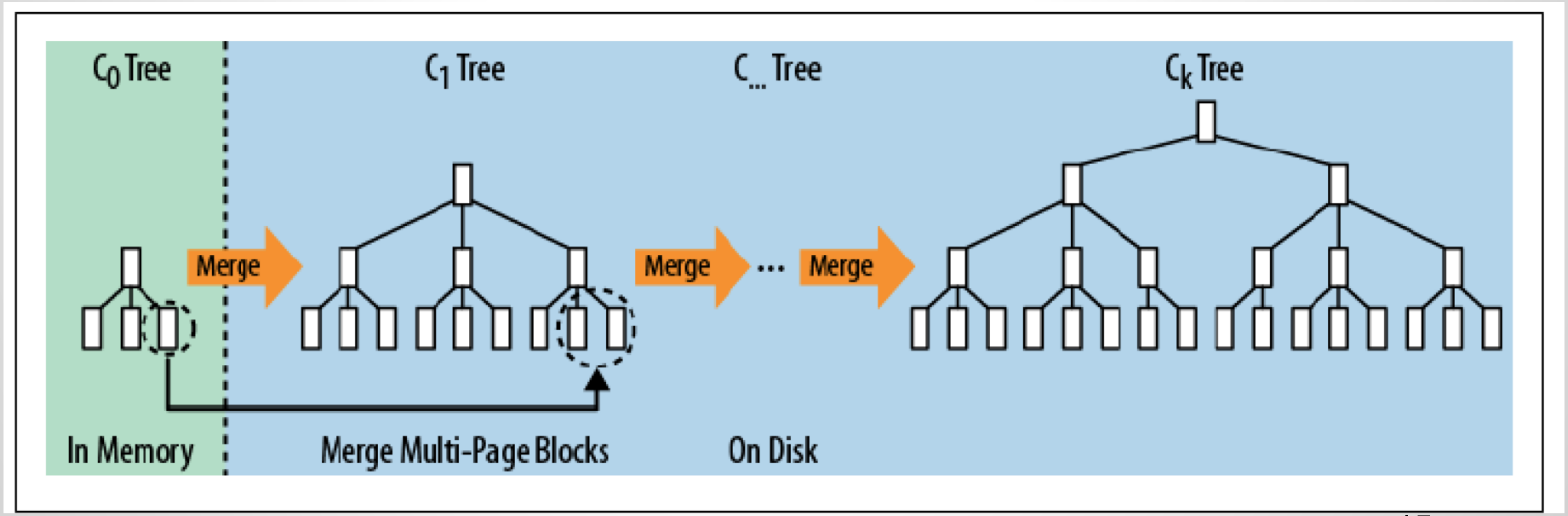
Interior Node
E
J
M
Leaf-level Nodes
A
B
C
D
E
F
G
H
I
J
K
L
d_A
d_B
d_C
d_D
d_E
d_F
d_G
d_H
d_I
d_J
d_K
d_L

Log-Structured Merge-Tree

- Log-structured merge-trees, also known as LSM-trees, follow a different approach. Incoming data is stored in a **logfile** first, completely **sequentially**. Once the log has the modification saved, it then updates an in-memory store that holds the most recent updates for fast lookup.
- When the system has accrued enough updates and starts to fill up the in-memory store, it flushes the sorted list of key → record pairs to disk, creating a new store file. Then the updates to the log can be thrown away, as all modifications have been persisted.

Log-Structured Merge-Tree

How a multipage block is merged from the in-memory tree into the next on-disk tree:



Compare: Seek VS Transfer

- B+ trees work well until there are too many modifications, because they force you to perform costly optimizations to retain that advantage for a limited amount of time. The more and faster you add data at random locations, the faster the pages become fragmented again. Eventually, you may take in data at a higher rate than the optimization process takes to rewrite the existing files. The updates and deletes are done at disk seek rates, rather than disk transfer rates.
- LSM-trees work at disk transfer rates and scale much better to handle large amounts of data. They also guarantee a very consistent insert rate, as they transform random writes into sequential writes using the logfile plus in-memory store.

Compare: Seek VS Transfer

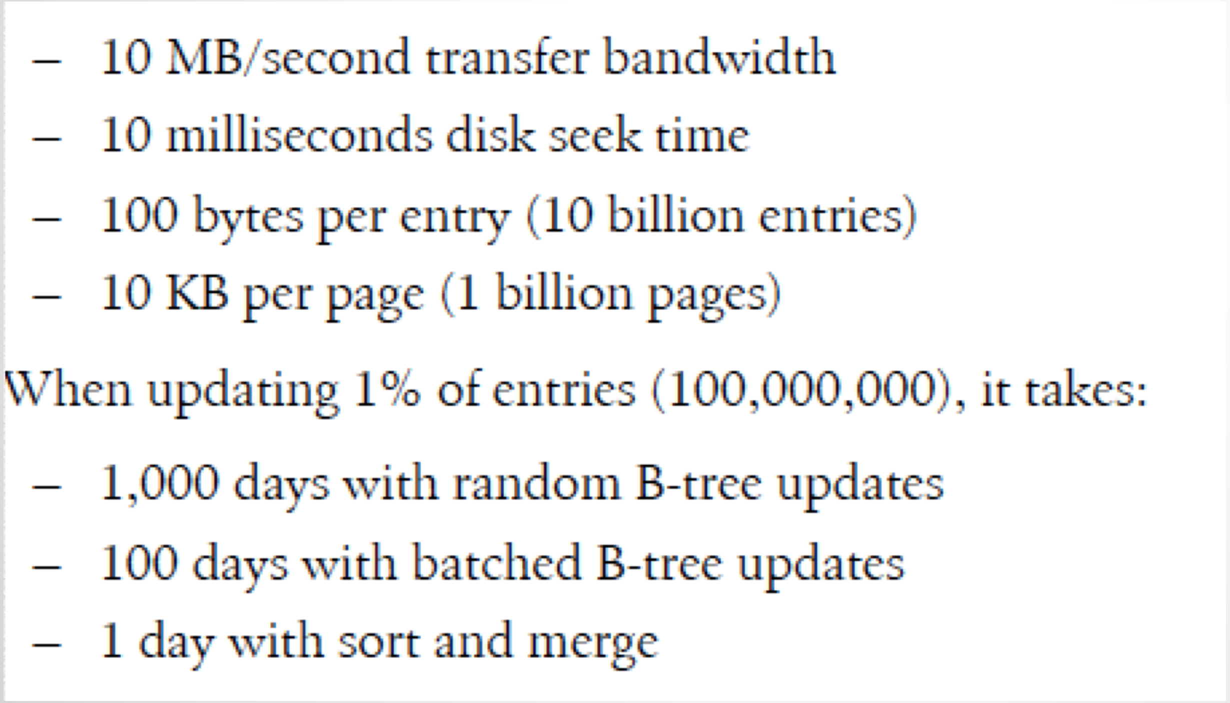
As discussed, there are two different database paradigms: one is seek and the other is transfer.

Seek is typically found in RDBMS and is caused by the B-tree or B+ tree structures used to store the data. It operates at the disk seek rate, resulting in log(N) seeks per access.

Transfer, on the other hand, as used by LSM-trees, sorts and merges files while operating at transfer rates, and takes log(updates) operations.

Compare: Seek VS Transfer

At scale seek, seek is inefficient compared to transfer:

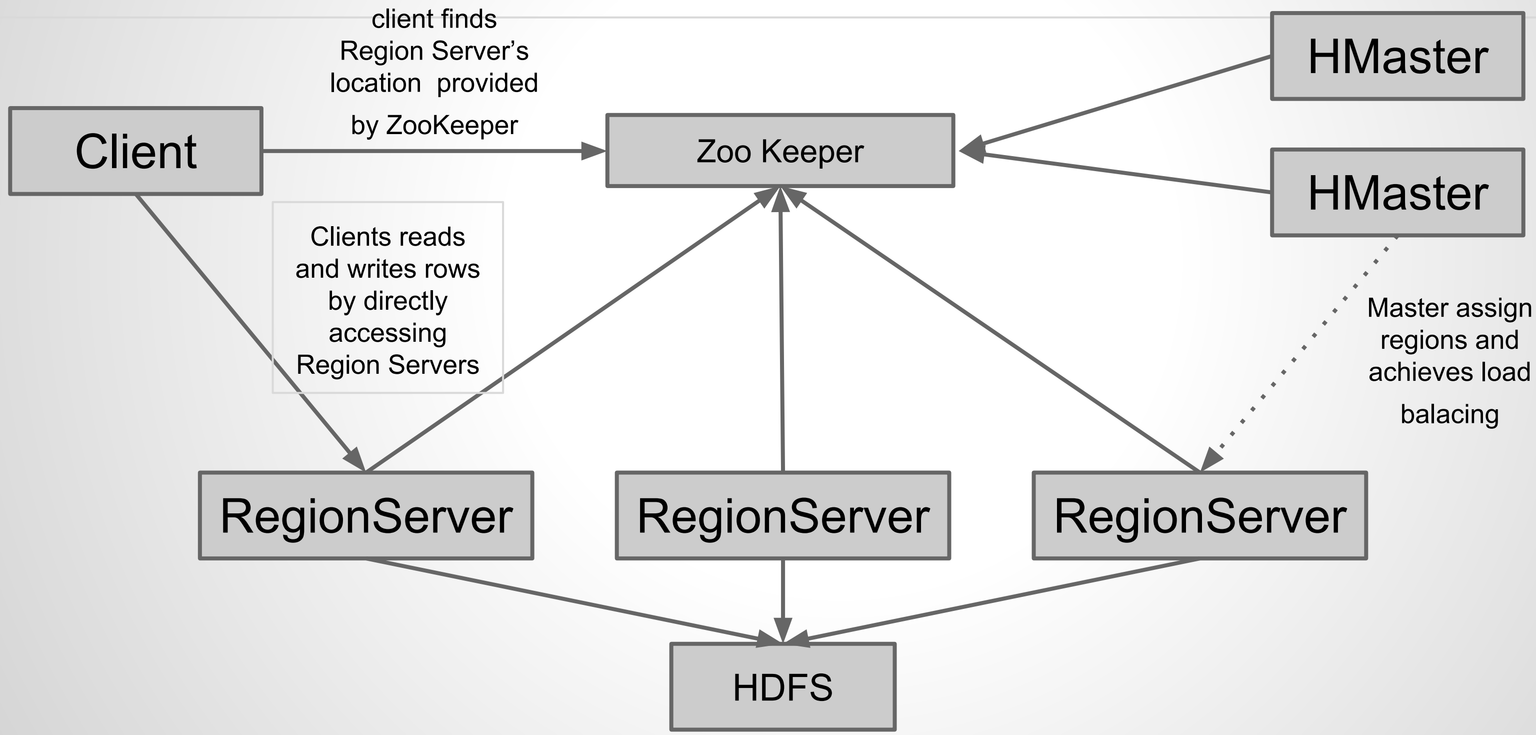


- 10 MB/second transfer bandwidth
- 10 milliseconds disk seek time
- 100 bytes per entry (10 billion entries)
- 10 KB per page (1 billion pages)

When updating 1% of entries (100,000,000), it takes:

- 1,000 days with random B-tree updates
- 100 days with batched B-tree updates
- 1 day with sort and merge

Cluster Architecture


client finds Region Server's location provided by ZooKeeper
Client
Zoo Keeper
HMaster
HMaster
Clients reads and writes rows by directly accessing Region Servers
Master assign regions and achieves load balacing
RegionServer
RegionServer
RegionServer
HDFS

-ROOT- and .META.

Zookeeper records the location of -ROOT- table
-ROOT- records Region information of .META. tables
.META. records Region information of user tables

The mapping of Regions to Region Server is kept in a system table called . META. When trying to read or write data from HBase, the clients read the required Region information from the .META table and directly communicate with the appropriate Region Server. Each Region is identified by the start key (inclusive) and the end key (exclusive)

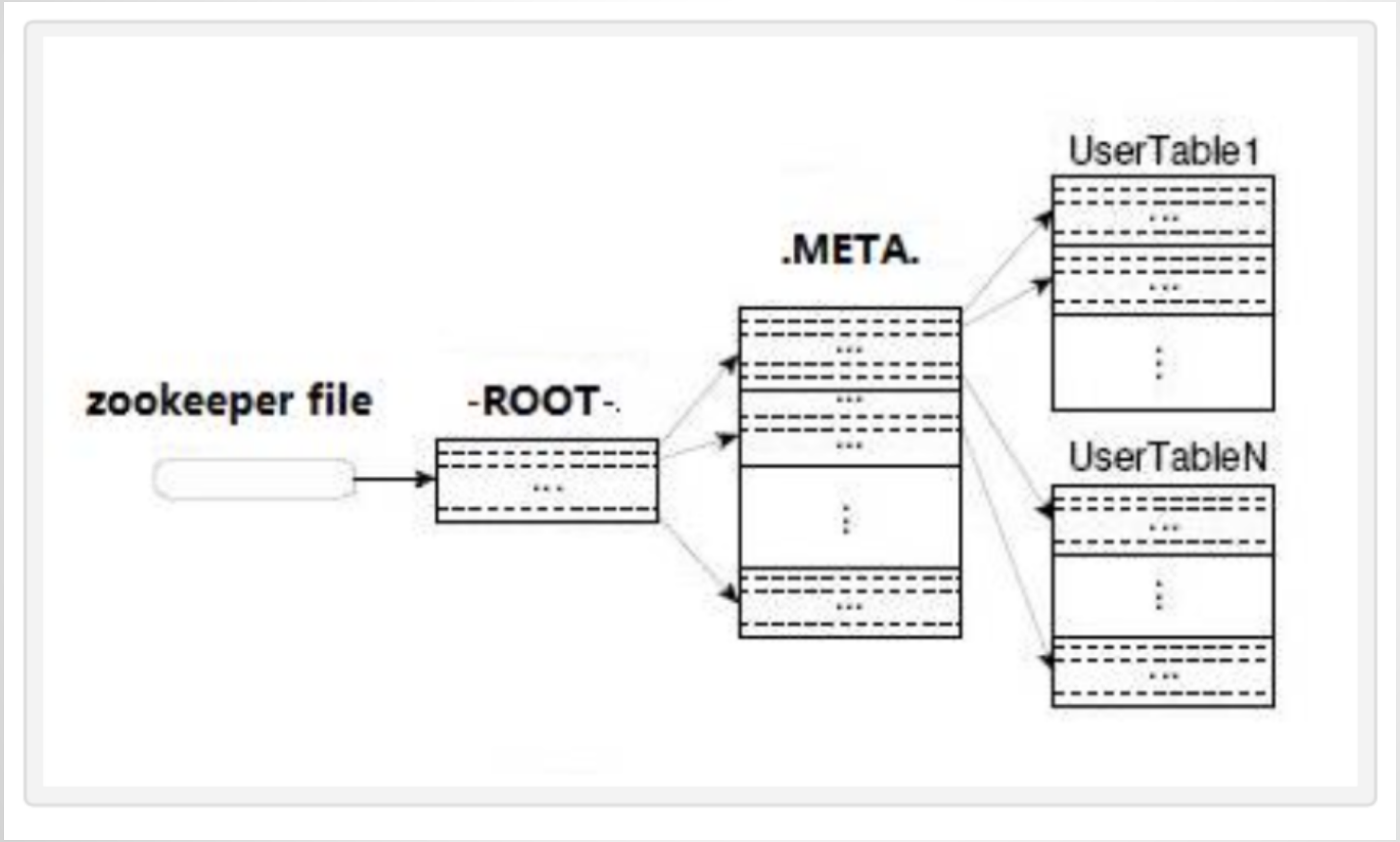
Communication Flow

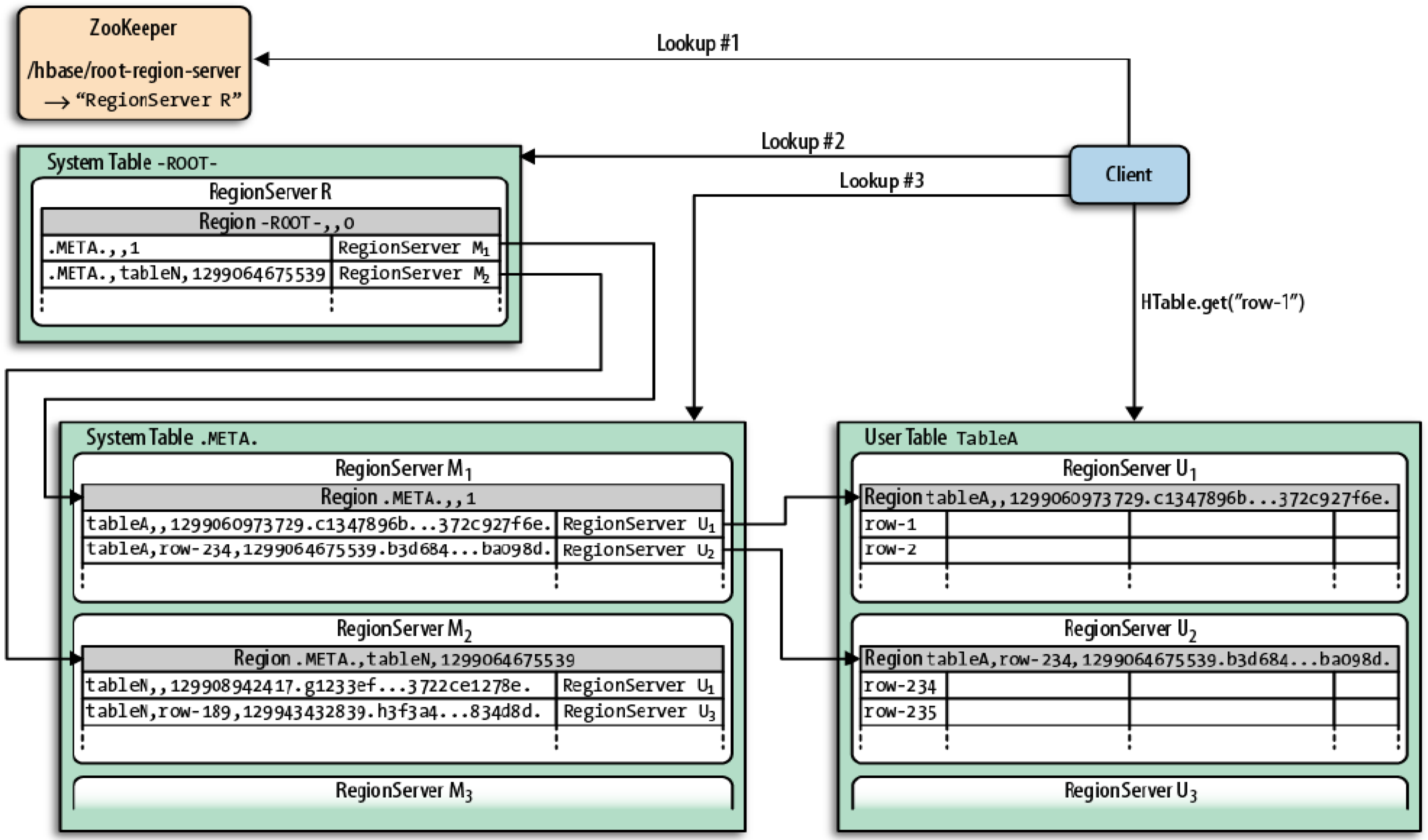
a new client contacts the ZooKeeper ensemble(a separate cluster of ZooKeeper nodes).It does so by retrieving the server name (i.e., hostname) that hosts the -ROOT- region from ZooKeeper.

query that region server to get the server name that hosts the .META. table region containing the row key.

query the reported .META. server and retrieve the server name that has the region containing the row key the client is looking for.

Communication Flow




ZooKeeper
/hbase/root-region-server
→ "RegionServer R"
Lookup #1
Lookup #2
Lookup #3
Client
HTable.get("row-1")
System Table -ROOT-
RegionServer R
Region -ROOT-,,0
.META.,,1
RegionServer M1
.META.,tableN,1299064675539
RegionServer M2
System Table .META.
RegionServer M1
Region .META.,,1
tableA,,1299060973729.c1347896b...372c927f6e.
RegionServer U1
tableA,row-234,1299064675539.b3d684...ba098d.
RegionServer U2
RegionServer M2
Region .META.,tableN,1299064675539
tableN,,129908942417.g1233ef...3722ce1278e.
RegionServer U1
tableN,row-189,129943432839.h3f3a4...834d8d.
RegionServer U3
RegionServer M3
User Table TableA
RegionServer U1
Region tableA,,1299060973729.c1347896b...372c927f6e.
row-1
row-2
RegionServer U2
Region tableA,row-234,1299064675539.b3d684...ba098d.
row-234
row-235
RegionServer U3

Summary

- Motivation -> Random read/write access
- Data Model -> Column family and qualifier, Versions
- Distributed Nature -> Master, Region, RegionServer
- Data Operations -> Get,Scan,Put,Delete
- Access APIs -> Java, REST, Thrift
- Architecture -> LSM Tree, Communication Workflow

Questions?

ZooKeeper

- ZooKeeper is a high-performance coordination service for distributed applications(like HBase). It exposes common services like naming, configuration management, synchronization, and group services, in a simple interface so you don't have to write them from scratch. You can use it off-the-shelf to implement consensus, group management, leader election, and presence protocols. And you can build on it for your own, specific needs.

- HBase relies completely on Zookeeper. HBase provides you the option to use its built-in Zookeeper which will get started whenever you start HBase.

- HBase depends on a running ZooKeeper cluster. All participating nodes and clients need to be able to access the running ZooKeeper ensemble. Apache HBase by default manages a ZooKeeper "cluster" for you. It will start and stop the ZooKeeper ensemble as part of the HBase start/stop process.