Apache HBase: the Hadoop Database

Yuanru Qian, Andrew Sharp, Jiuling Wang

Today we will discuss Apache HBase, the Hadoop Database. HBase is designed specifically for use by Hadoop, and we will define Hadoop soon, but first...

Agenda

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

Our agenda...

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.

Tells the story why we need HBase.

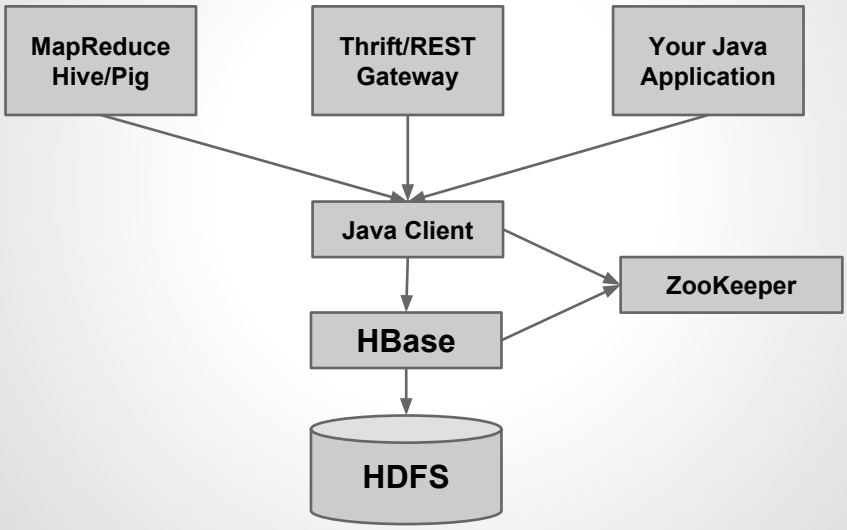
Introduction

- 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.

What is HBase?

Reference NoSql group's presentation on BigTable

Big Picture



Here's where Apache HBase fits into the Hadoop architecture. Here we can see Hadoop broken into a number of modules, but it's best to simply think of Hadoop as a large set of jobs to be completed over a large cluster. Each of these jobs needs data input to operate on and a data sink to place its output; HBase serves both of these needs. HBase uses HDFS, the Hadoop FileSystem, for writing to files that are distributed among a large cluster of computers. For the purposes of this lecture, it is unnecessary to go into great detail on HDFS.

Zookeeper is another largely unnecessary detail. It is sufficient to understand that it gives a client the address of the data it needs.

An Example Operation

The Job:

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

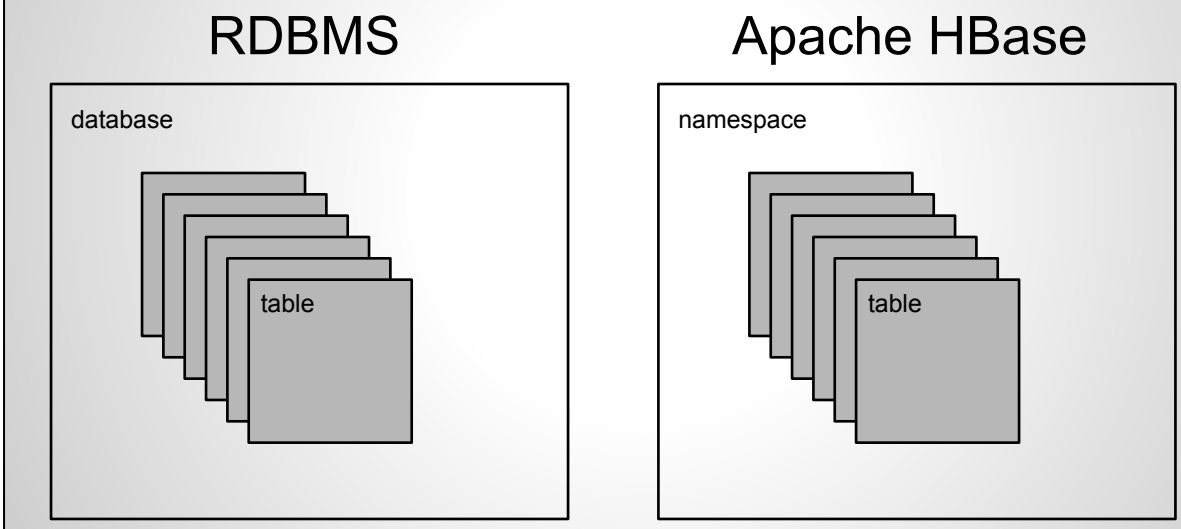
The Table:

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

An example use case.

The HBase Data Model

Data Model, Groups of Tables



Now we'll discuss the unique way that HBase stores its data. At a high level, it works very similar to a typical relation database machine. HBase organizes its tables into groups called namespaces. It is safe to see namespaces as no different than the databases that we used for Berkeley DB.

Data Model, Single Table

RDBMS

table	col1	col2	col3	col4
row1				
row2				
row3				

This is what we're used to seeing for a table.

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				

To start, HBase introduces *Column Families*, which you can see highlighted in blue. A Column Family is a group of columns. There are 2 primary advantages to grouping columns into families. Firstly, when defining a schema in HBase, you only need to define the Column Families of a table. You are free to add columns on the fly. This means that different rows can have different columns, and allows for high performance on sparse tables.

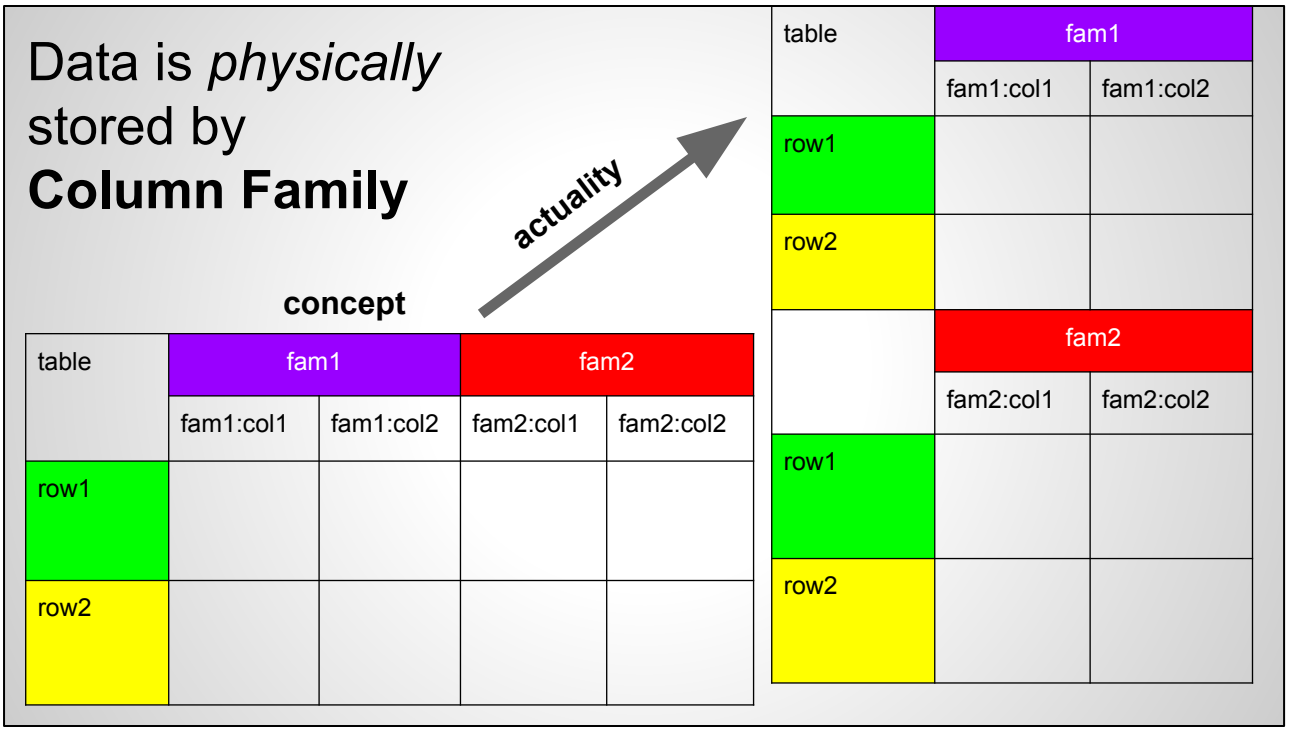
Columns – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

Column Families – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it's important that proper care be taken when designing Column Families in table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

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"

Here we show a sparse table as an example. You can see that not every row has a value for every column. Since HBase allows us to define columns per-row, we can avoid wasting space in this situation.



The second, and most important, advantage to having Column Families is the way they impact how data is stored via HBase. We are familiar with the strategy shown on the left, where data is stored first by row and then by column. If a database has few columns, this works fine for locality. However, HBase is designed to handle thousands of columns. If usage patterns indicate that most user operations only need a few columns from each row, it is inefficient to scan *all* of a row's columns for data. This is where Column Families come in. HBase stores data first by column family, then by row, then by column, as seen on the right. If a user designs their schema intelligently, they will put columns that are used in conjunction into the same column family. That way only the necessary column data is read. This represents an improvement to locality.

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		

This advantage to having Column Families persists after our data is sharded by row key. Since a table is sharded into regions which are then distributed across regionservers, if a full table scan is necessary for a subset of its columns, this design allows for high parallelism and high performance.

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				

The last bit of extra that HBase adds to its tables is *Versions*. It can hold up to 3 versions of data for each cell ((row, column) pair).

Version – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3. Version is a long integer.

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				

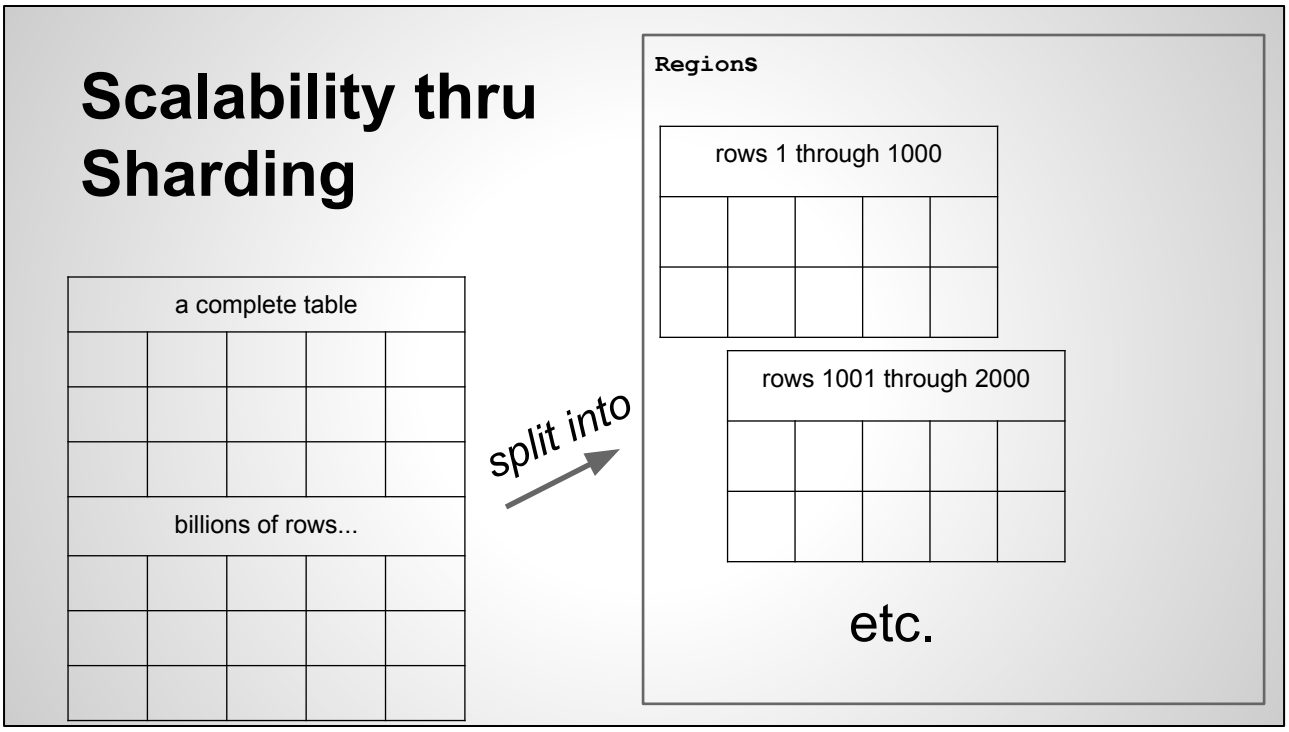
Self-explanatory.

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.

Review of all terms.

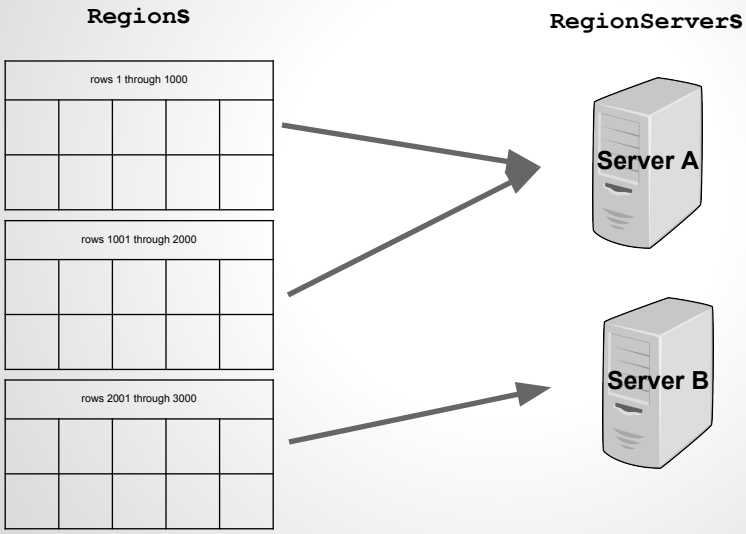
HBase's Distributed System



We've discussed sharding in this class, we should all be somewhat familiar with it. HBase implements sharding and relies heavily upon it for high performance. HBase implements sharding by splitting complete tables by row range into smaller pieces. In HBase parlance, we call these pieces "Regions".

Recall that a shard of a database includes all column families for that row range.

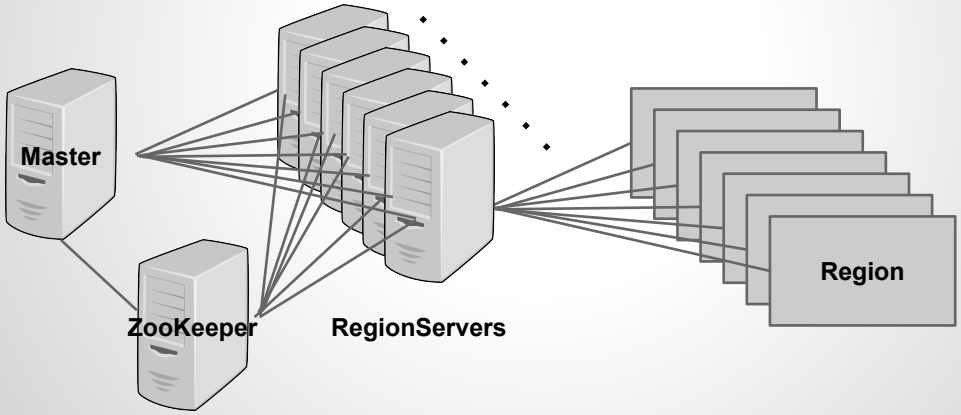
Scalability thru Sharding



Each “Region” is assigned to some machine. We call such a machine a “RegionServer”. Any number of regions can be assigned to any RegionServer. However, it should be noted that the more spread-out a table is sharded across RegionServers, the higher performance will be gained for a full-table scan.

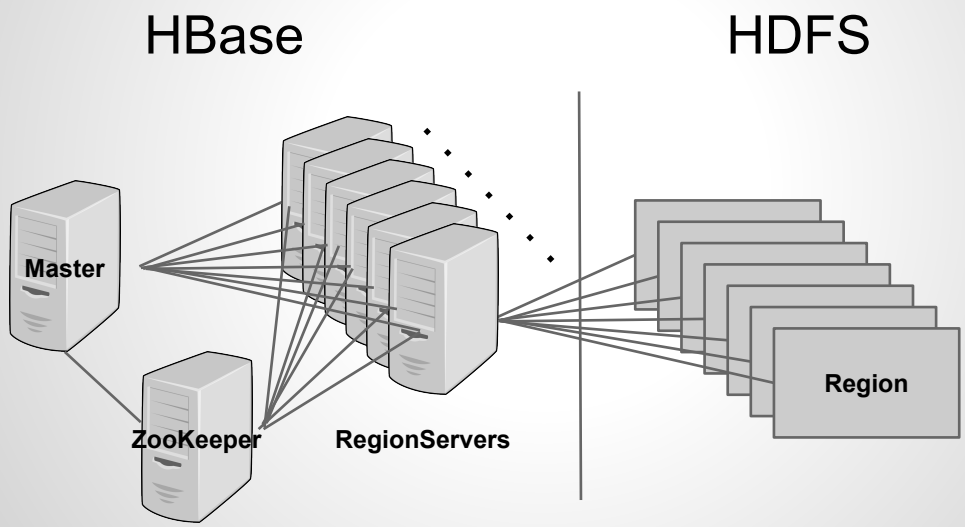
Scalability thru Division of Labor

An HBase Distributed System



We've already shown that a RegionServer is a machine that is responsible for some number of regions. If you want to access or update some piece of data, you simply contact the appropriate RegionServer and perform the operation. However, the first time that you need data from a certain Region, you need to know which RegionServer to contact. This is the purpose of the ZooKeeper machine.

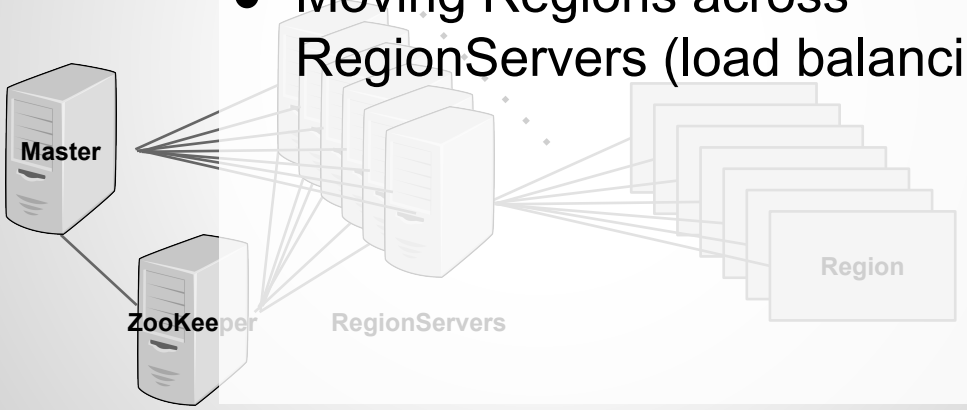
Scalability thru Division of Labor



A quick note about the relation between HBase and HDFS: HBase RegionServers do not actually hold data, they are just responsible for getting to it. The data is held by HDFS, which is responsible for such tasks as replication.

Division of Labor, Master

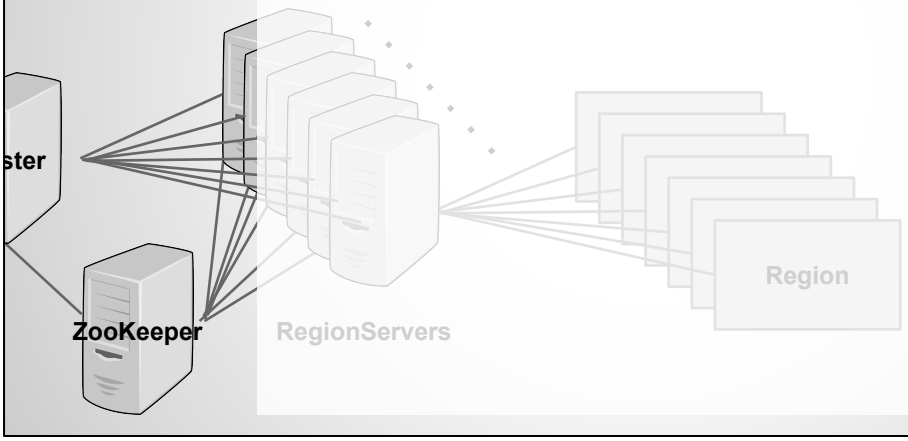
- Schema changes
- Moving Regions across RegionServers (load balancing)



The Master is contacted during any schema change, and is responsible for load-balancing, in which Regions are moved across RegionServers.

Division of Labor, ZooKeeper

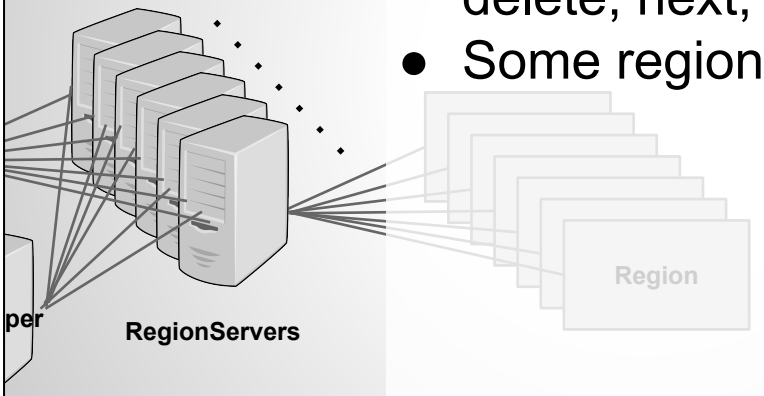
- Locating Regions



The ZooKeeper is contacted the first time a client accesses data, in order to get its address. After that, the address is cached.

Division of Labor, RegionServer

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



We've already discussed RegionServers a good deal, but we'd like to specify their role one more time. Each RegionServer is responsible for some number of regions. If you want to interact with data, you must talk to the appropriate RegionServer. Also, if a Region grows too large (through a series of insert commands to that Region), a RegionServer can split the Region on its own and maintain the two Regions that resulted from the split. However, if it begins to have too many Regions, it must contact the master to move a Region to another, less-busy RegionServer.

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

A final review of the terminology just discussed.

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.

HBase also achieves a good availability for different components.

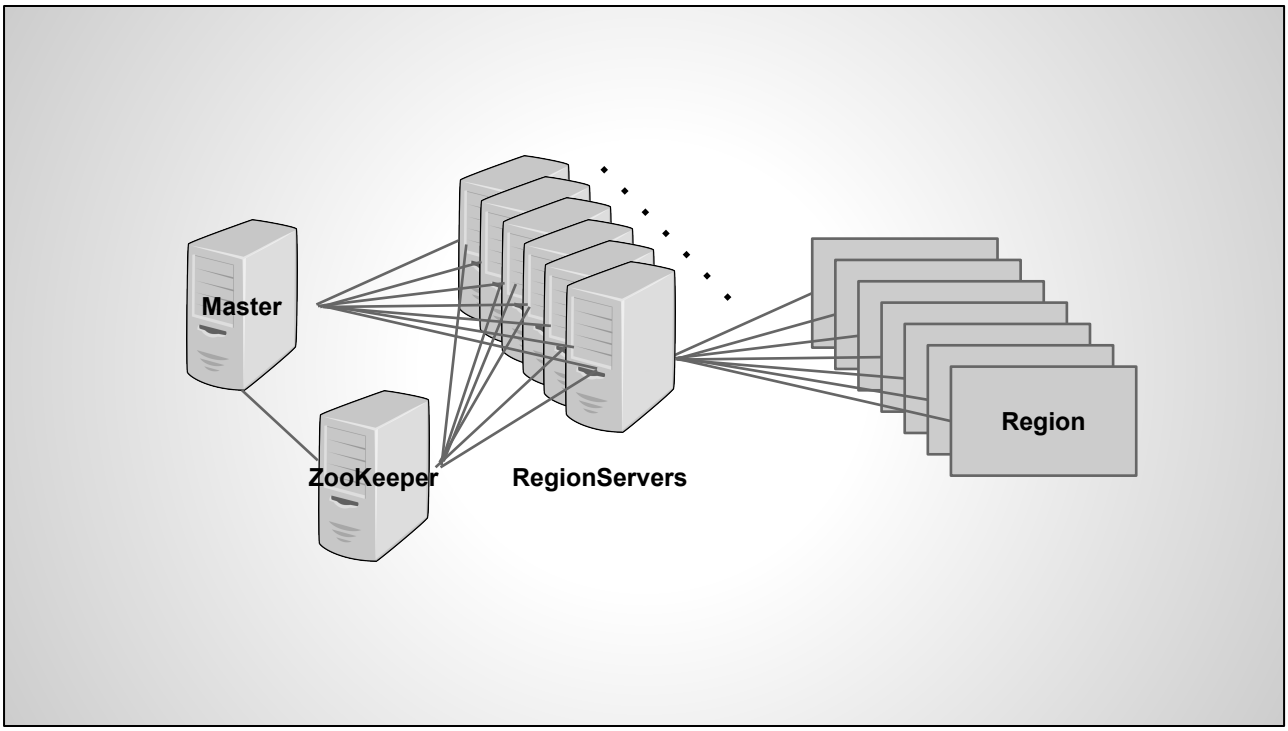


illustration of the failover for data nodes, region servers and master.

How to Access to HBase?

Java is the native language to implement HBase and thus is the native API for calling operations.

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).

Similar to the use of Berkely DB when you set up the environment.

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 columns called cf:qualifier from row 1 to row 1000.

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();
}
```

scan column family called cf from row 1 to row 1000.

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);
```

By executing different methods in get class, you can get an entire row, or a column family called “cf” of row1, or a column called cf:qualifier of row1, and in particular, the version2 of column family of row1.

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);
```

1. delete an entire row
2. delete the latest version of row1's column cf:qualifier
3. delete a particular version of row1's column cf:qualifier

Put

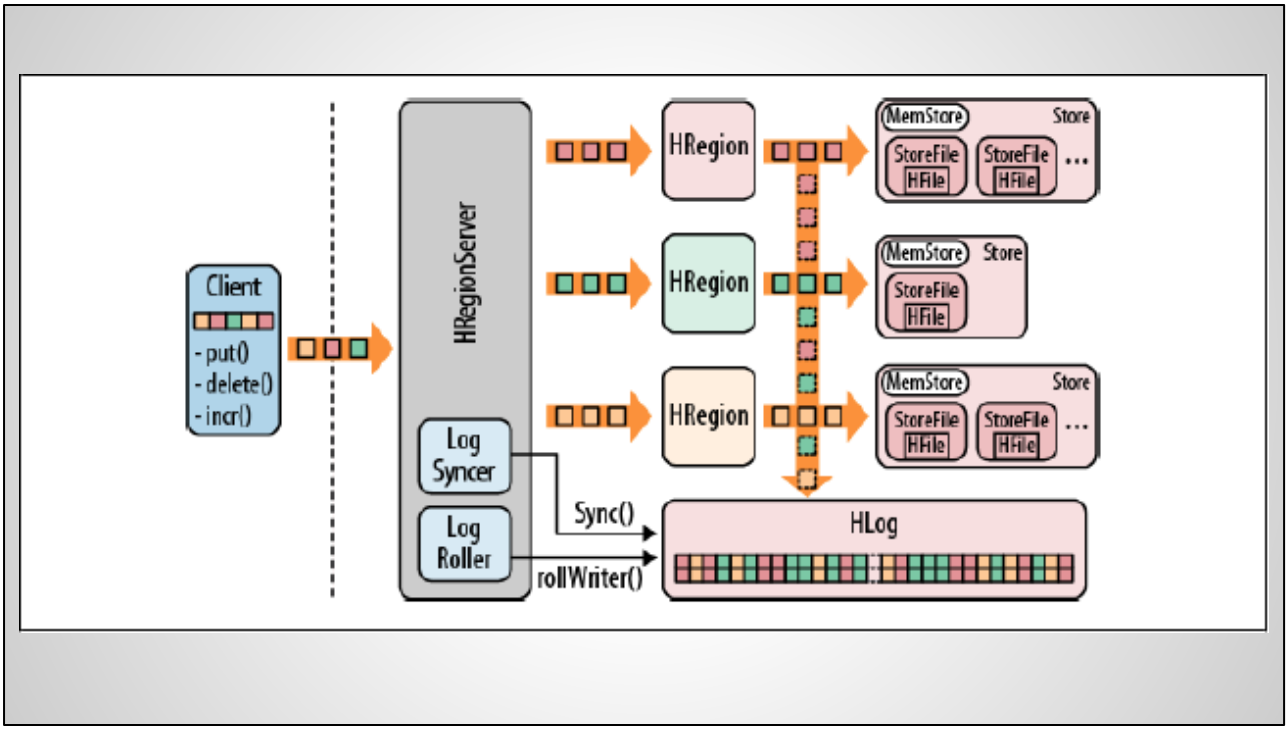
Put a new version of a cell using current timestamp by default

```
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);
```

1. if not specified, put add a new version
2. overwriting an existing value if version also stores.



The region server keeps data in-memory until enough is collected to warrant a flush to disk, avoiding the creation of too many very small files.

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.

How the method is implemented in a low level of the system?
If the server crashes, the WAL can effectively replay the log to get everything up to where the server should have been just before the crash. It also means if writing the record to the WAL fails, the whole operation is considered as a failure.
It shows the high availability of Hbase.

The KeyValue object contains the data as well as the coordinates of one specific cell. The coordinates are the row key, name of the column family,column qualifier, and timestamp.

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/...

In practice, SQL is based on joins and related low-level issues like foreign keys. SQL entices people to normalize their data. Normalization fragments databases into smaller tables which is great for data integrity and beneficial for some transactional systems. However, joins are expensive. Moreover, joins require strong consistency and fixed schemas.

In turn, avoiding join operations makes it possible to maintain flexible or informal schemas, and to scale horizontally. Thus, the NoSQL solutions should really be called NoJoin because they are mostly defined by avoidance of the join operation.

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 defines the semantics so that the protocol can be used in a generic way to address remote resource. By not changing the protocol, REST is compatible with existing technologies, such as web servers, and proxies. Resources are uniquely specified as part of the request URI.

A **network gateway** is an *internetworking* system capable of joining together two networks that use different base protocols.

REST

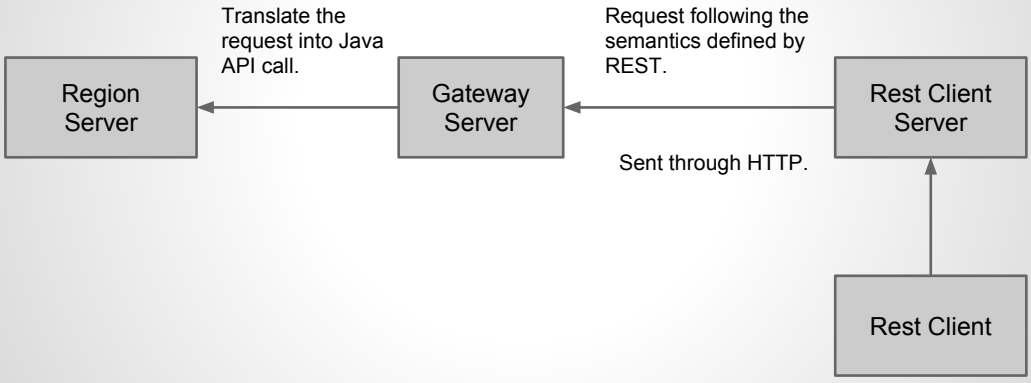
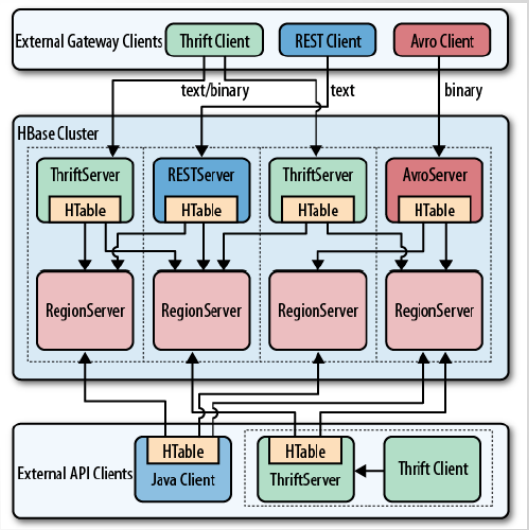


Illustration example.

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



For this image, you will see two approaches of clients interacting with HBase Cluster.

External Gateway Clients: on top of the region server processes, sharing the same physical machine. There is no true recommendation for how to place the gateway servers. You may way to collocate them, or have them on dedicated machines.

External API Clients: run ThriftServer or REST servers directly on the client nodes. For example, when you have web servers constructing the resultant HTML pages using PHP, it is better to run the gateway process on the same server. That way, the communication between the client and the gateway is local, while the RPC between the gateway and HBase is using the native protocol.

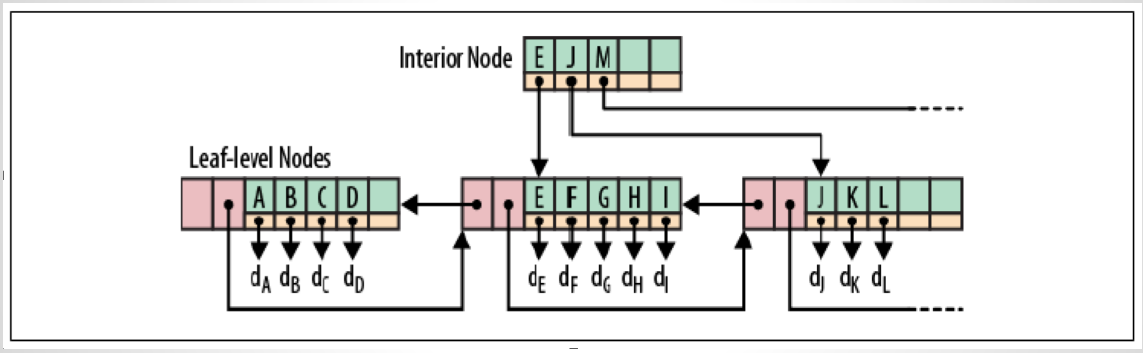
REST client -> client server -> gateway server -> invoke RPC to HBase Cluster-> return data

Architecture

storage structures:

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

B+ Trees



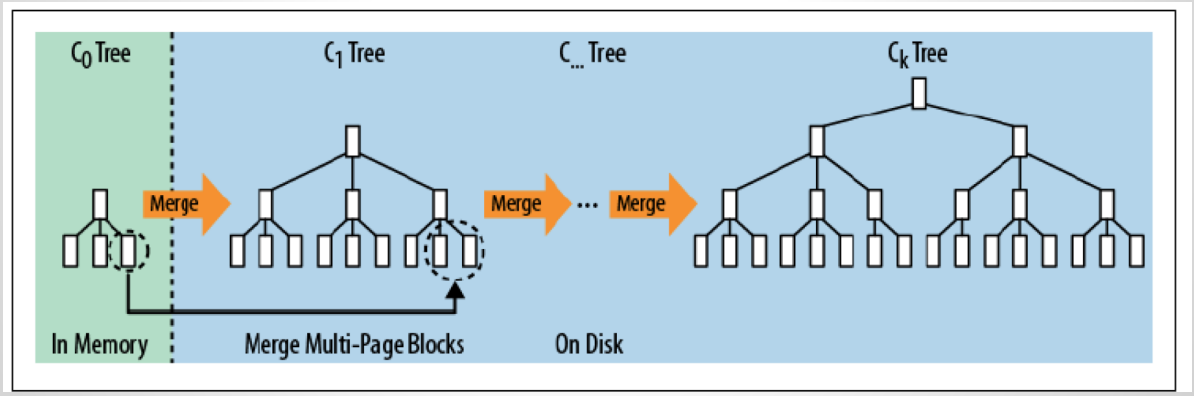
same as what we've learned, won't talk too much

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:



The store files are arranged similar to B-trees, but are optimized for sequential disk access where all nodes are completely filled and stored as either single-page or multipage blocks. Updating the store files is done in a rolling merge fashion, that is, the system packs existing on-disk multipage blocks together with the flushed in-memory data until the block reaches its full capacity, at which point a new one is started.

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:

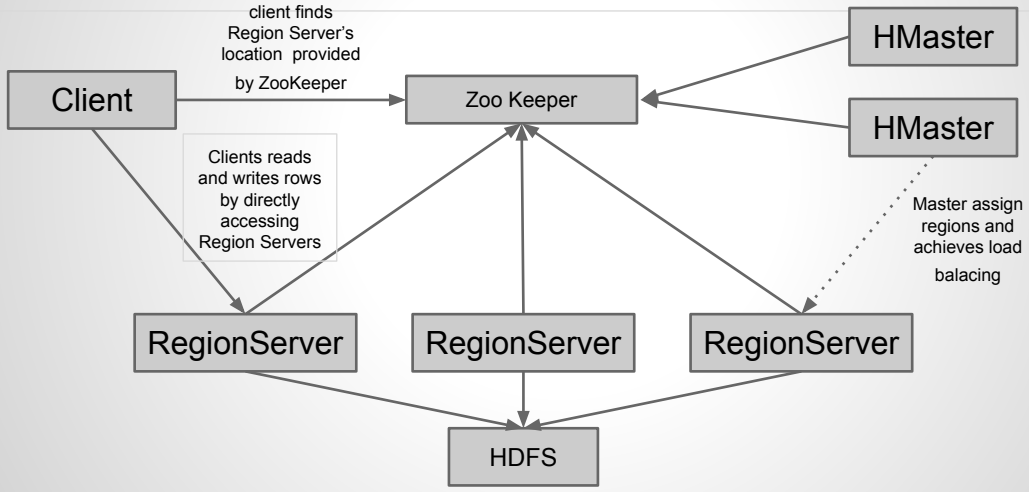
- 10 MB/second transfer bandwidth
- 10 milliseconds disk seek time
- 100 bytes per entry (10 billion entries)
- 10 KB per page (1 billion pages)

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

Seek Versus Sort and Merge in Numbers
batched updates:
When a large number of new keys are to be inserted (or deleted) into a B-tree at about the same time, it is often profitable to sort the keys in the main memory before performing updates to the B-tree on disk. Thus, updates falling into the same leaf of the B-tree can be performed simultaneously and disk accesses are saved.

Cluster Architecture



This is a more complex diagram of the distributed architecture of HBase. We won't go into detail about ZooKeeper.

-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)

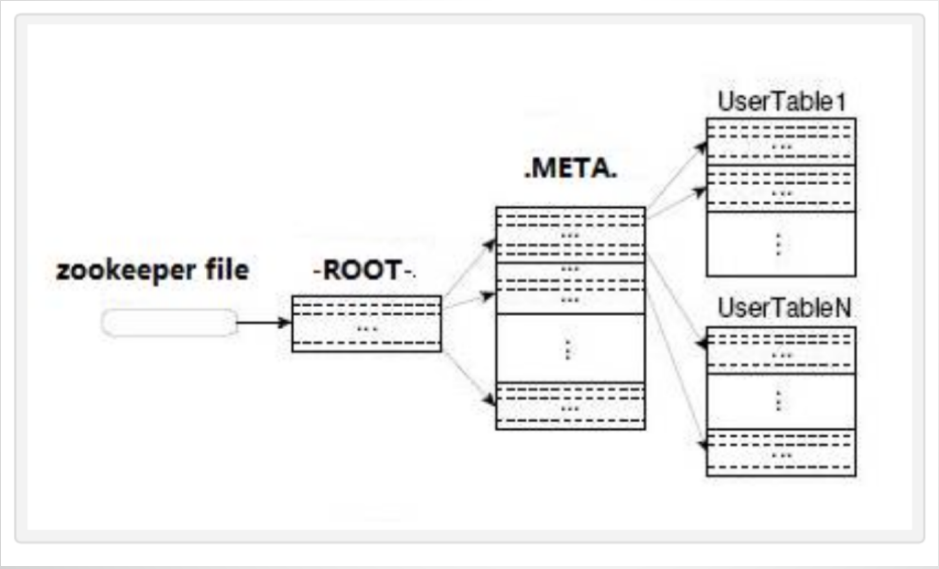
Communication Flow

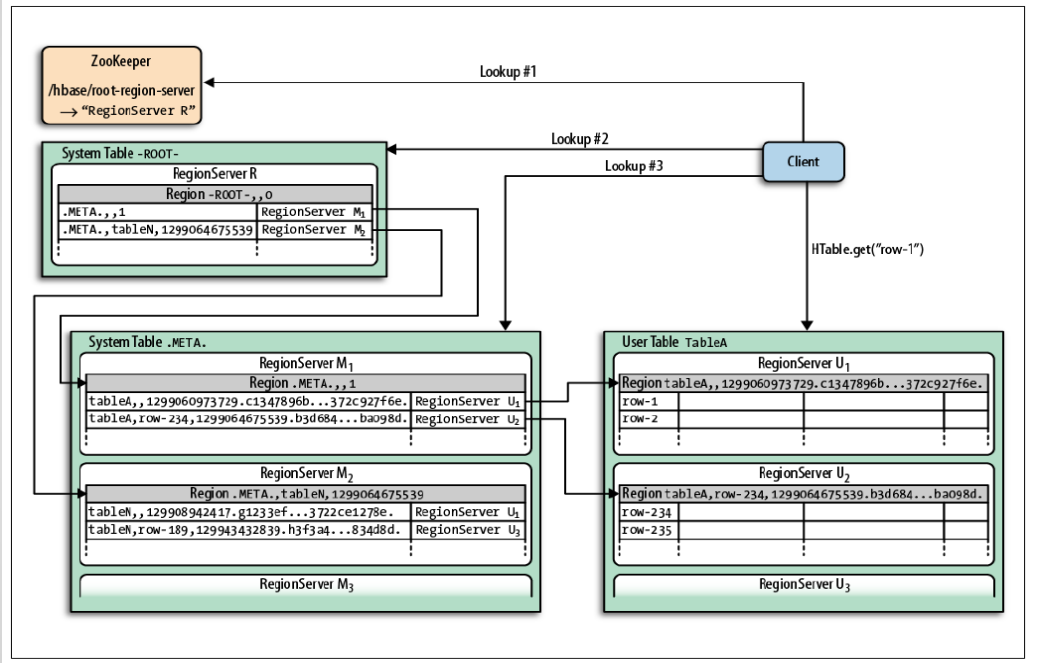
a new client contacts the ZooKeeper ensemble(a separate cluster of ZooKeeper nodes).It does so by retrieving the server name (i.e., hostname) that hosts the -ROOT- region from ZooKeeper.

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

example

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

Feature
Random, real-time read/write access to data
Linear scalability to store hundreds of TB of data
Automatic and configurable sharing of tables
Strictly consistent reads and writes
High availability through automatic failover

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.