

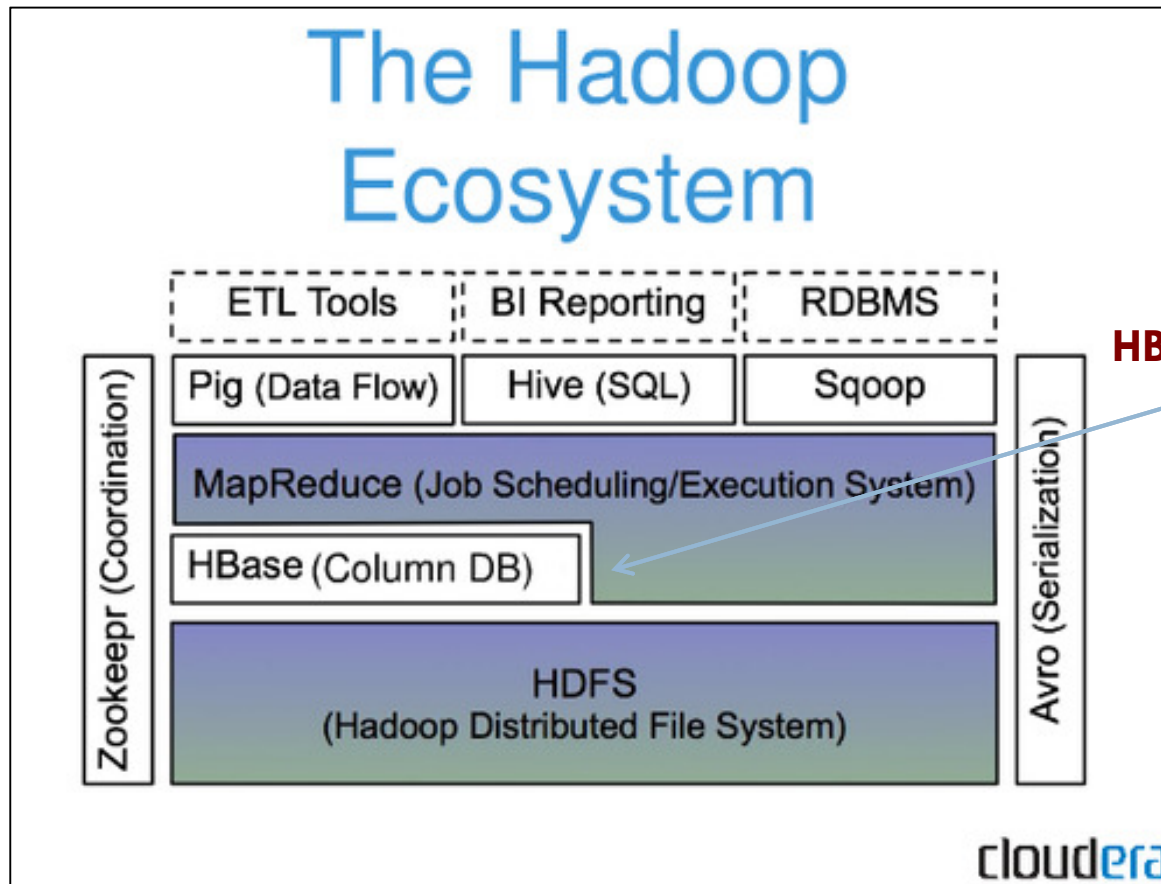
# HBase: Overview

1

- HBase is a distributed column-oriented data store built on top of HDFS
- HBase is an Apache open source project whose goal is to provide storage for the Hadoop Distributed Computing
- Data is logically organized into tables, rows and columns

# HBase: Part of Hadoop's Ecosystem

2



**HBase is built on top of HDFS**



**HBase files are internally stored in HDFS**

# HBase vs. HDFS

3

- Both are distributed systems that scale to hundreds or thousands of nodes
- HDFS is good for batch processing (scans over big files)
  - ▣ Not good for record lookup
  - ▣ Not good for incremental addition of small batches
  - ▣ Not good for updates

# HBase vs. HDFS (Cont'd)

4

- HBase is designed to efficiently address the above points
  - ▣ Fast record lookup
  - ▣ Support for record-level insertion
  - ▣ Support for updates (not in place)
- HBase updates are done by creating new versions of values

# HBase vs. HDFS (Cont'd)

5

	<b>Plain HDFS/MR</b>	<b>HBase</b>
<b>Write pattern</b>	<b>Append-only</b>	<b>Random write, bulk incremental</b>
<b>Read pattern</b>	<b>Full table scan, partition table scan</b>	<b>Random read, small range scan, or table scan</b>
<b>Hive (SQL) performance</b>	<b>Very good</b>	<b>4-5x slower</b>
<b>Structured storage</b>	<b>Do-it-yourself / TSV / SequenceFile / Avro / ?</b>	<b>Sparse column-family data model</b>
<b>Max data size</b>	<b>30+ PB</b>	<b>~1PB</b>

If application has neither random reads or writes → Stick to HDFS

# HBase Run Modes

6

- HBase run modes:
  - ▣ Standalone
  - ▣ Distributed
- Out of the box, HBase runs in standalone mode.
- Whatever your mode, you will need to configure HBase by editing files in the HBase conf directory.
- At a minimum, you must edit `conf/hbase-env.sh` to tell HBase which java to use.
- In this file you set HBase environment variables such as the heapsize and other options for the JVM, the preferred location for log files, etc.
- Set `JAVA_HOME` to point at the root of your java install.

# Standalone Mode

7

- This is the default mode.
- In standalone mode, HBase does not use HDFS -- it uses the local filesystem instead -- and it runs all HBase daemons and a local ZooKeeper all up in the same JVM.
- Zookeeper binds to a well known port so clients may talk to HBase.

# Local Filesystem and Durability

8

- ❑ Using HBase with a LocalFileSystem does not currently guarantee durability.
- ❑ The HDFS local filesystem implementation will lose edits if files are not properly closed -- which is very likely to happen when experimenting with a new download.
- ❑ You need to run HBase on HDFS to ensure all writes are preserved.
- ❑ Running against the local filesystem though will get you off the ground quickly and get you familiar with how the general system works.
- ❑ See <https://issues.apache.org/jira/browse/HBASE-3696> and its associated issues for more details.



# After Unpacking Hbase, Edit conf/hbase-site.xml

9

- Set hbase.rootdir, the directory HBase writes data to, and hbase.zookeeper.property.dataDir, the directory ZooKeeper writes its data too:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///DIRECTORY/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/DIRECTORY/zookeeper</value>
  </property>
</configuration>
```
- Replace DIRECTORY in the above with the path to the directory you would have HBase and ZooKeeper write their data.
- By default, hbase.rootdir is set to /tmp/hbase-`{user.name}` and similarly so for the default ZooKeeper data location which means you'll lose all your data after reboot unless you change it.

# Start HBase

10

- Now start HBase:

```
$ ./bin/start-hbase.sh
```

starting Master, logging to logs/hbase-user-master-example.org.out

- You should now have a running standalone HBase instance.
- In standalone mode, HBase runs all daemons in the the one JVM; i.e. both the HBase and ZooKeeper daemons.
- HBase logs can be found in the logs subdirectory.
- Check them out especially if it seems HBase had trouble starting.

# Connect to your running HBase via the shell.

11

- ❑ `./bin/hbase shell`  
HBase Shell; enter 'help<RETURN>' for list of supported commands.  
Type "exit<RETURN>" to leave the HBase Shell  
`hbase(main):001:0>`
- ❑ Type `help` and then `<RETURN>` to see a listing of shell commands and options.
- ❑ Create a table named `test` with a single column family named `cf`.
- ❑ Verify its creation by listing all tables and then insert some values.  
`hbase(main):003:0> create 'test', 'cf'`  
`0 row(s) in 1.2200 seconds`  
`hbase(main):003:0> list 'test'`  
`..`  
`1 row(s) in 0.0550 seconds`  
`hbase(main):004:0> put 'test', 'row1', 'cf:a', 'value1'`  
`0 row(s) in 0.0560 seconds`  
`hbase(main):005:0> put 'test', 'row2', 'cf:b', 'value2'`  
`0 row(s) in 0.0370 seconds`  
`hbase(main):006:0> put 'test', 'row3', 'cf:c', 'value3'`  
`0 row(s) in 0.0450 seconds`

# Scanning Tables

12

- Above we inserted 3 values, one at a time. The first insert is at row1, column cf:a with a value of value1.
- Columns in HBase are comprised of a column family prefix -- cf in this example -- followed by a colon and then a column qualifier suffix.
- Verify the data insert by running a scan of the table as follows

```
hbase(main):007:0> scan 'test'
ROW          COLUMN+CELL
row1         column=cf:a, timestamp=1288380727188, value=value1
row2         column=cf:b, timestamp=1288380738440, value=value2
row3         column=cf:c, timestamp=1288380747365, value=value3
3 row(s) in 0.0590 seconds
```

- Get a single row

```
hbase(main):008:0> get 'test', 'row1'
COLUMN      CELL
cf:a        timestamp=1288380727188, value=value1
1 row(s) in 0.0400 seconds
```

- Now, disable and drop your table. This will clean up all done above.

```
hbase(main):012:0> disable 'test'
0 row(s) in 1.0930 seconds
hbase(main):013:0> drop 'test'
0 row(s) in 0.0770 seconds
```

# Stopping HBase

13

- Stop your hbase instance by running the stop script.

```
$ ./bin/stop-hbase.sh
```

```
stopping hbase.....
```

# Distributed Mode

14

- Distributed mode can be subdivided into
  - ▣ pseudo-distributed – distributed but all daemons run on a single node
  - ▣ fully-distributed – where the daemons are spread across all nodes in the cluster.
- Pseudo-distributed mode can run against the local filesystem or it can run against an instance of the Hadoop Distributed File System (HDFS).
- Fully-distributed mode can ONLY run on HDFS.

# Pseudo-Distributed

15

- A pseudo-distributed mode is simply a fully-distributed mode run on a single host.
- Use this configuration testing and prototyping on HBase.
- Do not use this configuration for production nor for evaluating HBase performance.
- If you want to run on HDFS rather than on the local filesystem, setup your HDFS.
- Ensure you have a working HDFS before proceeding.

# Configuring HBase for Pseudo-Distributed Mode

16

- ❑ Configure HBase. Edit conf/hbase-site.xml.
- ❑ This is the file into which you add local customizations and overrides.
- ❑ At a minimum, you must tell HBase to run in (pseudo-)distributed mode rather than in default standalone mode.
- ❑ To do this, set the hbase.cluster.distributed property to true (Its default is false).
- ❑ The absolute bare-minimum hbase-site.xml is therefore as follows:

```
<configuration>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
</configuration>
```

- ❑ With this configuration, HBase will start up an HBase Master process, a ZooKeeper server, and a RegionServer process running against the local filesystem writing to wherever your operating system stores temporary files into a directory named:
  - ▣ hbase-YOUR\_USER\_NAME.



# Using HDFS

17

- Such a setup, using the local filesystem and writing to the operating systems's temporary directory is an ephemeral setup; the Hadoop local filesystem -- which is what HBase uses when it is writing the local filesystem -- would lose data unless the system was shutdown properly in versions of HBase before 0.98.4 and 1.0.0 (see HBASE-11218 Data loss in HBase standalone mode).
- Writing to the operating system's temporary directory can also make for data loss when the machine is restarted as this directory is usually cleared on reboot.
- For a more permanent setup, where we make use of an instance of HDFS; HBase data will be written to the Hadoop distributed filesystem rather than to the local filesystem's tmp directory.
- Below, the hbase.rootdir property points to the local HDFS instance homed on the node h-24-30.my-host.com.
- Let HBase create the hbase.rootdir directory. If you don't, you'll get warning saying HBase needs a migration run because the directory is missing files expected by HBase (it'll create them if you let it).

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://h-24-30.my-host.com:8020/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
</configuration>
```

# Starting Initial HBase Cluster

18

- To start up the initial HBase cluster...

```
% bin/start-hbase.sh
```

- To start up an extra backup master(s) on the same server run...

```
% bin/local-master-backup.sh start 1
```

- ... the '1' means use ports 16001 & 16011, and this backup master's logfile will be at logs/hbase-\${USER}-1-master-\${HOSTNAME}.log.

- To startup multiple backup masters run...

```
% bin/local-master-backup.sh start 2 3
```

- You can start up to 9 backup masters (10 total).

- To start up more regionserver...

```
% bin/local-regionserver.sh start 1
```

- ... where '1' means use ports 16201 & 16301 and its logfile will be at logs/hbase-\${USER}-1-regionserver-\${HOSTNAME}.log.

- To add 4 more regionserver in addition to the one you just started by running...

```
% bin/local-regionserver.sh start 2 3 4 5
```

- This supports up to 99 extra regionserver (100 total).

# Fully-Distributed Mode

19

- For running a fully-distributed operation on more than one host, make the following configurations.
- In `hbase-site.xml`, add the property `hbase.cluster.distributed` and set it to `true` and point the HBase `hbase.rootdir` at the appropriate HDFS NameNode and location in HDFS where you would like HBase to write data. For example, if you namenode were running at `namenode.example.org` on port 8020 and you wanted to home your HBase in HDFS at `/hbase`, make the following configuration.

```
<configuration>
```

```
...
```

```
<property>
```

```
  <name>hbase.rootdir</name>
```

```
  <value>hdfs://namenode.example.org:8020/hbase</value>
```

```
  <description>The directory shared by RegionServers.
```

```
  </description>
```

```
</property>
```

```
<property>
```

```
  <name>hbase.cluster.distributed</name>
```

```
  <value>true</value>
```

```
  <description>The mode the cluster will be in. Possible values are
```

```
    false: standalone and pseudo-distributed setups with managed Zookeeper
```

```
    true: fully-distributed with unmanaged Zookeeper Quorum (see hbase-env.sh)
```

```
  </description>
```

```
</property>
```

# HDFS Client Configuration

20

- Of note, if you have made HDFS client configuration on your Hadoop cluster -- i.e. configuration you want HDFS clients to use as opposed to server-side configurations -- HBase will not see this configuration unless you do one of the following:
  - ▣ Add a pointer to your HADOOP\_CONF\_DIR to the HBASE\_CLASSPATH environment variable in hbase-env.sh.
  - ▣ Add a copy of hdfs-site.xml (or hadoop-site.xml) or, better, symlinks, under \${HBASE\_HOME}/conf, or
  - ▣ if only a small set of HDFS client configurations, add them to hbase-site.xml.
- An example of such an HDFS client configuration is dfs.replication.
- If for example, you want to run with a replication factor of 5, hbase will create files with the default of 3 unless you do the above to make the configuration available to HBase.

# Running and Confirming Your Installation

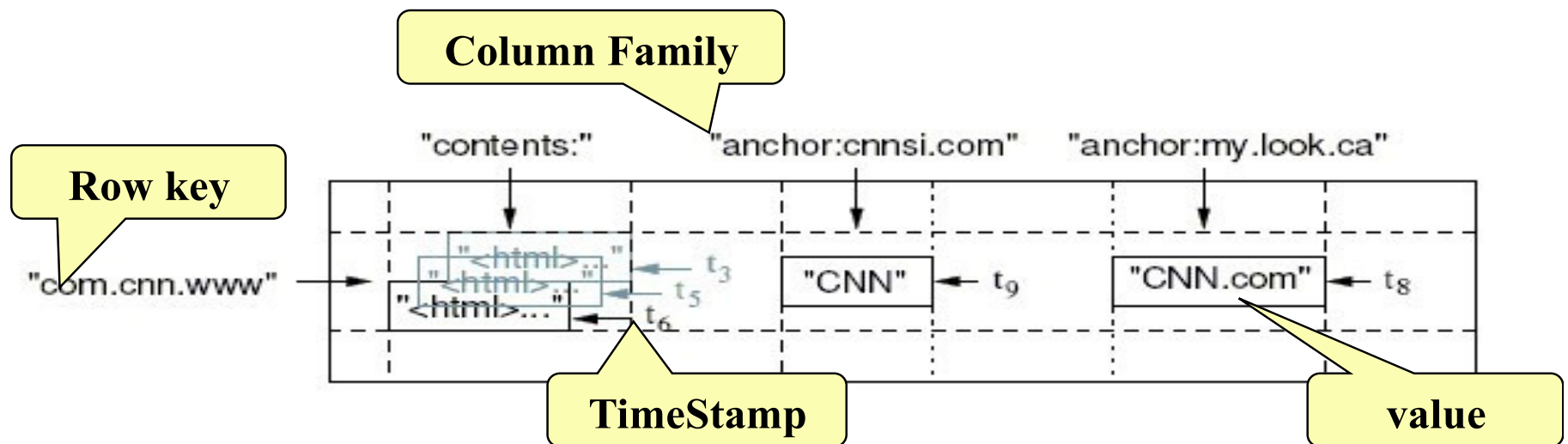
21

- ❑ Make sure HDFS is running first.
- ❑ Start and stop the Hadoop HDFS daemons by running `bin/start-hdfs.sh` over in the `HADOOP_HOME` directory.
- ❑ You can ensure it started properly by testing the put and get of files into the Hadoop filesystem.\
- ❑ HBase does not normally use the mapreduce daemons. These do not need to be started.
- ❑ If you are managing your own ZooKeeper, start it and confirm its running else, HBase will start up ZooKeeper for you as part of its start process.
- ❑ Start HBase with the following command:  
`bin/start-hbase.sh`
- ❑ Run the above from the `HBASE_HOME` directory.
- ❑ You should now have a running HBase instance.
- ❑ HBase logs can be found in the logs subdirectory. Check them out especially if HBase had trouble starting.
- ❑ HBase also puts up a UI listing vital attributes. By default its deployed on the Master host at port 16010 (HBase RegionServers listen on port 16020 by default and put up an informational http server at 16030). If the Master were running on a host named `master.example.org` on the default port, to see the Master's homepage you'd point your browser at `http://master.example.org:16010`.
- ❑ Prior to HBase 0.98, the default ports the master ui was deployed on port 16010, and the HBase RegionServers would listen on port 16020 by default and put up an informational http server at 16030.

# HBase Data Model

22

- HBase is based on Google's Bigtable model
  - ▣ Key-Value pairs



# HBase Logical View

23

Implicit PRIMARY KEY in  
RDBMS terms

Data is all `byte[]` in HBase

Different types of  
data separated into  
different  
“column families”

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7', 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

Different rows may have different sets  
of columns(table is *sparse*)

A single cell might have different  
values at different timestamps

Useful for \*-To-Many mappings

# HBase: Keys and Column Families

24

Each row has a **Key**

Each record is divided into **Column Families**

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**



### □ Key

- ▣ Byte array
- ▣ Serves as the primary key for the table
- ▣ Indexed for fast lookup

### □ Column Family

- ▣ Has a name (string)
- ▣ Contains one or more related columns

### □ Column

- ▣ Belongs to one column family
- ▣ Included inside the row
  - **familyName:columnName**

Column family named "Contents"

Column family named "anchor"

Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html>..."		
	t11	"<html>..."	Column named "apache.com"	
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnn.com"	"CNN"
	t13		"anchor:mylook.ca"	"CNN.com"
	t6	"<html>..."		
	t5	"<html>..."		
	t3	"<html>..."		

Version number for each row

### □ Version Number

- Unique within each key
- By default → System's timestamp
- Data type is Long

### □ Value (Cell)

- Byte array

Row key	Time Stamp	Column “content s:”	Column “anchor:”	
“com.apac he.ww w”	t12	“<html> ...”		value
	t11	“<html> ...”		
	t10		“anchor:apache .com”	“APACH E”
“com.cnn.w ww”	t15		“anchor:cnnsi.co m”	“CNN”
	t13		“anchor.my.look. ca”	“CNN.co m”
	t6	“<html> ...”		
	t5	“<html> ...”		
	t3	“<html> ...”		

# Notes on Data Model

27

- HBase schema consists of several **Tables**
- Each table consists of a set of **Column Families**
  - ▣ Columns are not part of the schema
- HBase has **Dynamic Columns**
  - ▣ Because column names are encoded inside the cells
  - ▣ Different cells can have different columns

“Roles” column family  
has different columns in  
different cells




Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7', 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

# Notes on Data Model (Cont'd)

- The **version number** can be user-supplied
  - Even does not have to be inserted in increasing order
  - Version number are unique within each key
- Table can be very sparse
  - Many cells are empty
- **Keys** are indexed as the primary key

Has two columns  
[cnnsi.com & my.look.ca]



Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"
"com.cnn.www"	t6	contents:html = "<html>..."	
"com.cnn.www"	t5	contents:html = "<html>..."	
"com.cnn.www"	t3	contents:html = "<html>..."	

# HBase Physical Model

29

- Each column family is stored in a separate file (called **HTables**)
- Key & Version numbers are replicated with each column family
- Empty cells are not stored

HBase maintains a multi-level index on values:

*<key, column family, column name, timestamp>*

**Table 5.3. ColumnFamily contents**

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

**Table 5.2. ColumnFamily anchor**

Row Key	Time Stamp	Column Family anchor
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

# Example

30

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7', 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

## info Column Family

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

## roles Column Family

Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted  
on disk by  
Row key, Col  
key,  
descending  
timestamp

Milliseconds since unix epoch



# Column Families

31

- Different sets of columns may have different properties and access patterns
- Configurable by column family:
  - Compression (none, gzip, LZO)
  - Version retention policies
  - Cache priority
- CFs stored separately on disk: access one without wasting IO on the other.

# HBase Regions

32

- Each HTable (column family) is partitioned horizontally into *regions*
  - ▣ Regions are counterpart to HDFS blocks

**Table 5.3. ColumnFamily contents**

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

*Each will be one region*

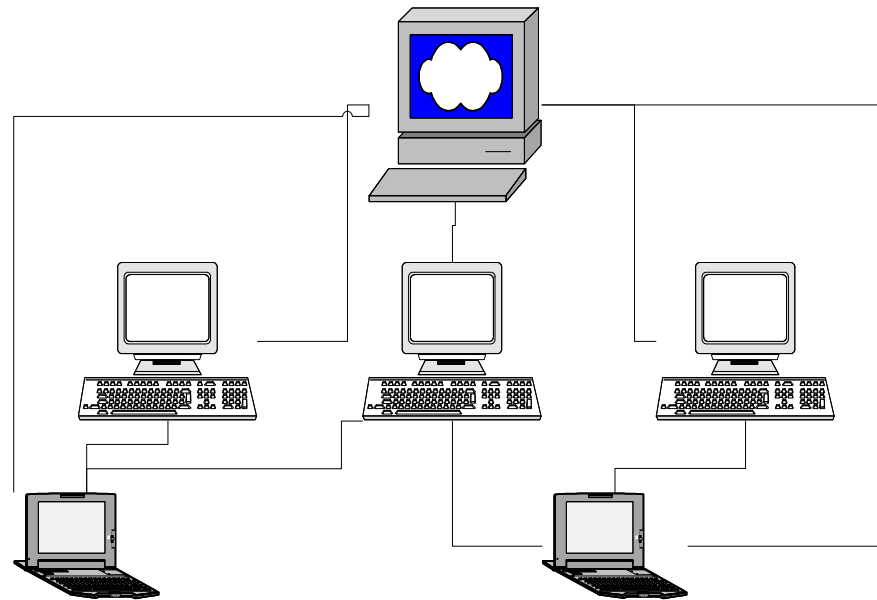


# HBase Architecture

# HBase Architecture – 3 Major Components

34

- The HBaseMaster
  - ▣ One master
- The HRegionServer
  - ▣ Many region servers
- The HBase client



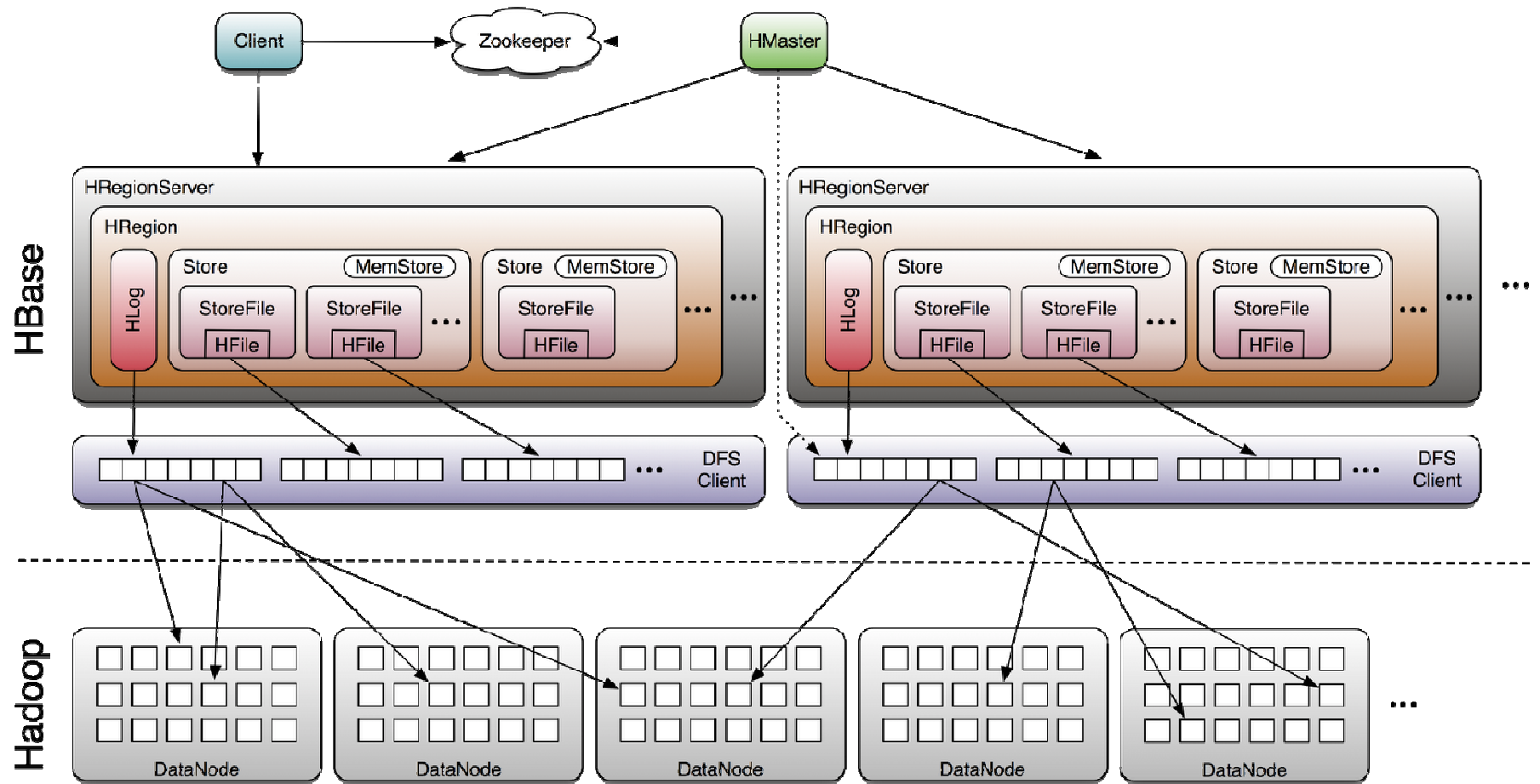
# HBase Components

35

- **Region**
  - ▣ A subset of a table's rows, like horizontal range partitioning
  - ▣ Automatically done
- **RegionServer (many slaves)**
  - ▣ Manages data regions
  - ▣ Serves data for reads and writes (*using a log*)
- **Master**
  - ▣ Responsible for coordinating the slaves
  - ▣ Assigns regions, detects failures
  - ▣ Admin functions

# Big Picture

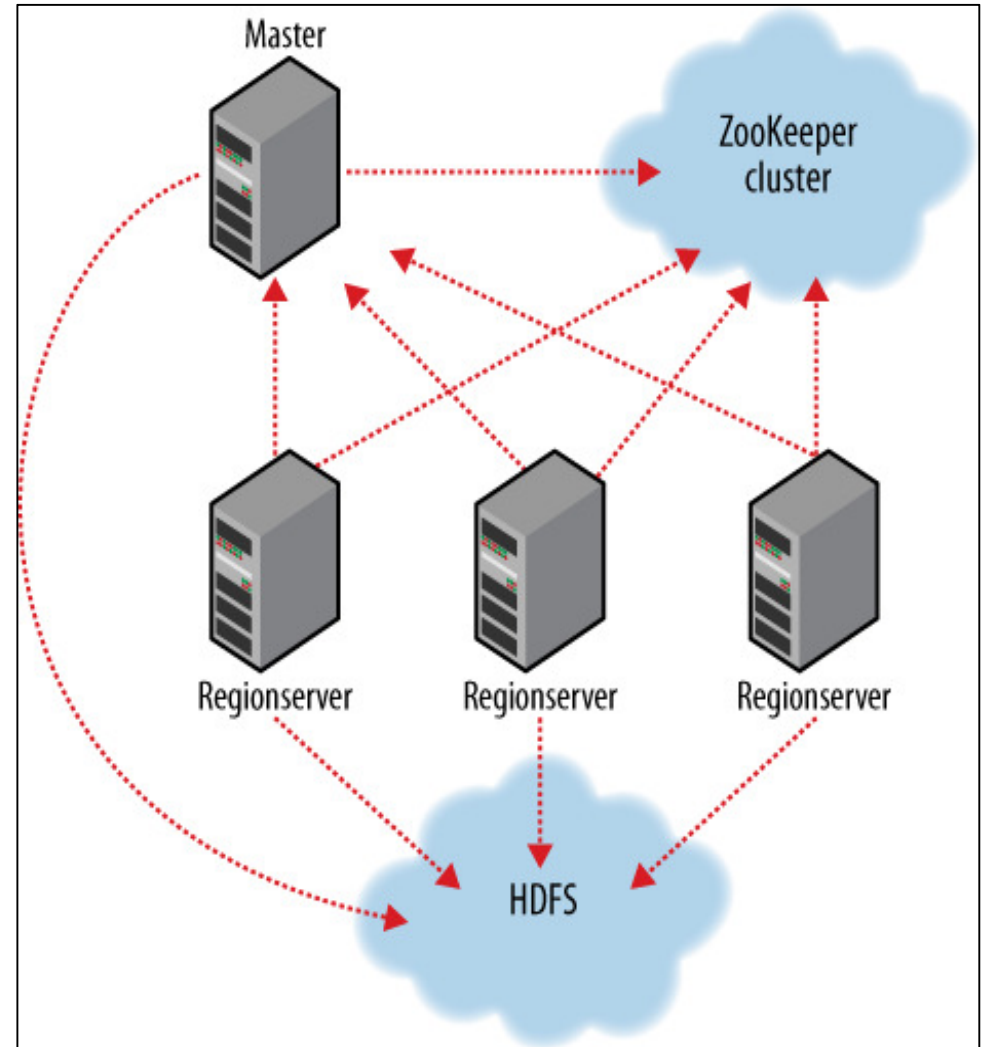
36



# ZooKeeper

37

- HBase depends on ZooKeeper
- By default HBase manages the ZooKeeper instance
  - ▣ E.g., starts and stops ZooKeeper
- HMaster and HRegionServers register themselves with ZooKeeper



# Creating a Table

38

```
HBaseAdmin admin = new HBaseAdmin(config);

HColumnDescriptor [] column;
column= new HColumnDescriptor[2];
column[0]=new HColumnDescriptor("columnFamily1:");
column[1]=new HColumnDescriptor("columnFamily2:");

HTableDescriptor desc;
desc = new HTableDescriptor(Bytes.toBytes("MyTable"));
desc.addFamily(column[0]);
desc.addFamily(column[1]);
admin.createTable(desc);
```

# Operations On Regions: **Get()**

39

- Given a key → return corresponding record
- For each value return the highest version
- The following Get will only retrieve the current version of the row

```
Get get = new Get(Bytes.toBytes("row1"));
```

```
Result r = htable.get(get);
```

```
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns current version of value
```

- Can control the number of versions you want
- The following Get will return the last 3 versions of the row.

```
Get get = new Get(Bytes.toBytes("row1"));
```

```
get.setMaxVersions(3); // will return last 3 versions of row
```

```
Result r = htable.get(get);
```

```
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns current version of value
```

```
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns all versions
```

# Operations On Regions: **Scan()**

It allows iteration over multiple rows for specified attributes

40

- Assume that a table is populated with rows with keys "row1", "row2", "row3", and then another set of rows with the keys "abc1", "abc2", and "abc3". The following example shows how startRow and stopRow can be applied to a Scan instance to return the rows beginning with "row".

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes();

...
HTable htable = ... // instantiate HTable
Scan scan = new Scan();
scan.addColumn(CF, ATTR);
scan.setStartRow(Bytes.toBytes("row")); // start key is inclusive
scan.setStopRow(Bytes.toBytes("rox")); // stop key is exclusive
ResultScanner rs = htable.getScanner(scan);
try {
    for (Result r = rs.next(); r != null; r = rs.next()) {
        // process result...
    } finally {
        rs.close(); // always close the ResultScanner!
    }
}
```



# Get()

Select value from table where  
key='com.apache.www' AND  
label='anchor:apache.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

# Scan()

Select value from table  
where anchor='cnnsi.com'

Row key	Time Stamp	Column "anchor:"	
"com.apache.www"	t12		
	t11		
	t10	"anchor:apache.com"	"APACHE"
"com.cnn.www"	t9	"anchor:cnnsi.com"	"CNN"
	t8	"anchor:my.look.ca"	"CNN.com"
	t6		
	t5		
	t3		

# Operations On Regions: Put()

43

- Insert a new record (with a new key), Or
- Insert a record for an existing key

**Implicit version number  
(timestamp)**

```
Put put = new Put(Bytes.toBytes(row));  
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.toBytes(data));  
htable.put(put);
```

**Explicit version number**

```
Put put = new Put(Bytes.toBytes(row));  
long explicitTimeInMs = 555; // just an example  
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), explicitTimeInMs, Bytes.toBytes(data));  
htable.put(put);
```

# Operations On Regions: Delete()

44

- Marking table cells as deleted
- **Multiple levels**
  - ▣ Can mark an entire column family as deleted
  - ▣ Can make all column families of a given row as deleted

- All operations are logged by the RegionServers
- The log is flushed periodically

# HBase: Joins

45

- HBase does not support joins
- Can be done in the application layer
  - ▣ Using scan() and get() operations

# Altering a Table

46

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(conf);
String table = "myTable";

admin.disableTable(table);

HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1);           // adding new ColumnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2);        // modifying existing ColumnFamily

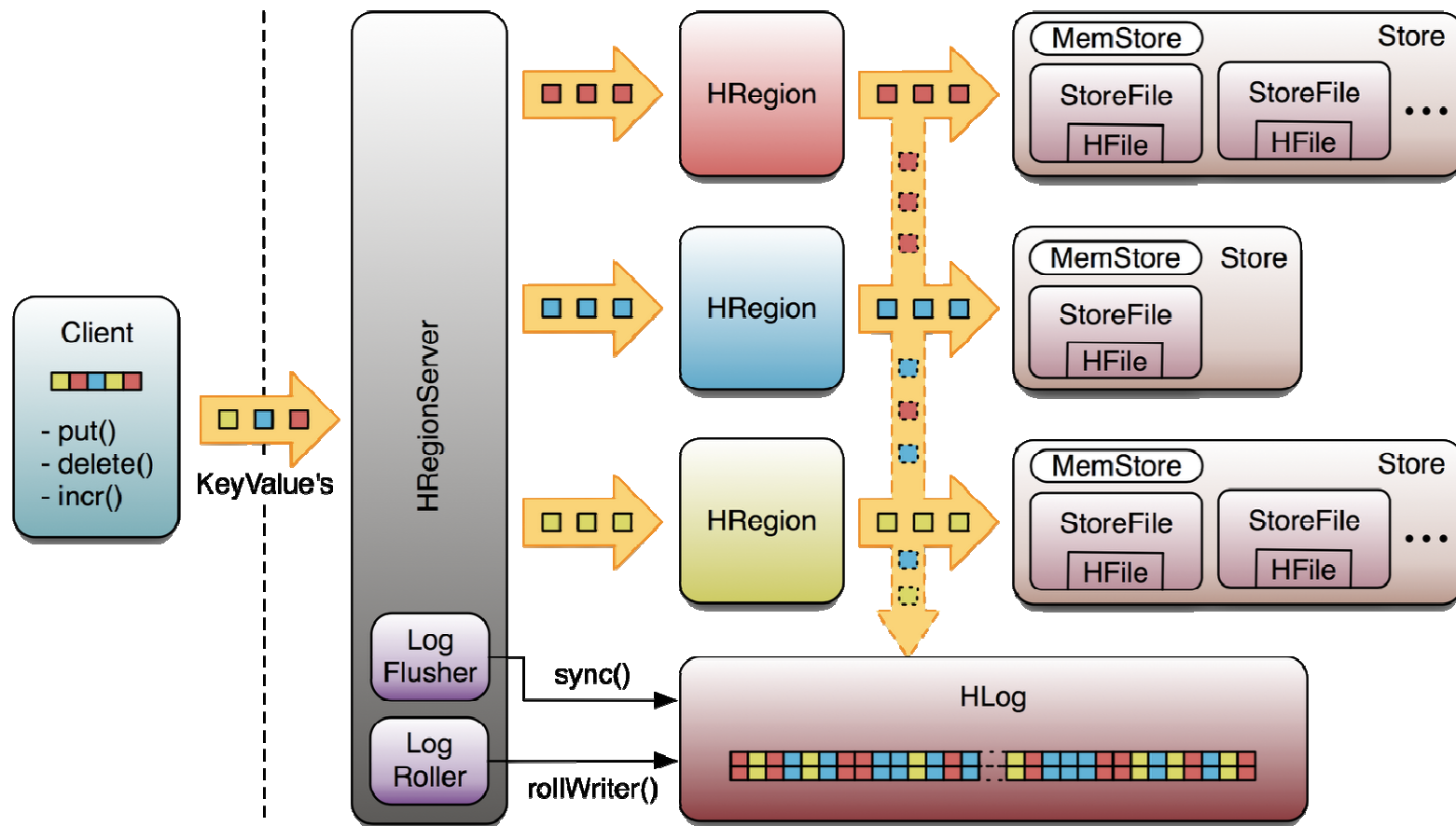
admin.enableTable(table);
```

Disable the table before changing the schema

6.1. Schema Creation

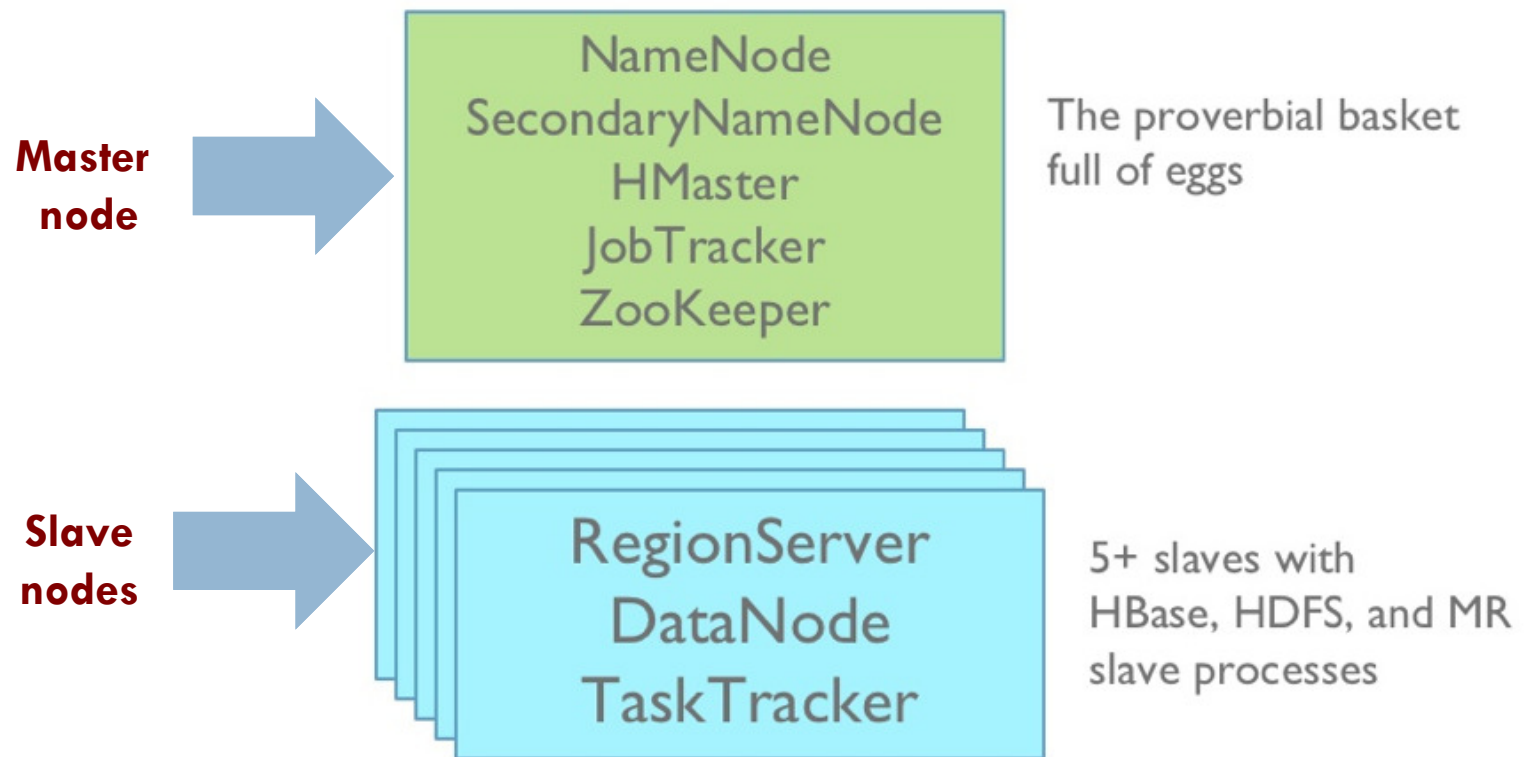
# Logging Operations

47



# HBase Deployment

48





# HBase vs. HDFS

49

	<b>Plain HDFS/MR</b>	<b>HBase</b>
<b>Write pattern</b>	<b>Append-only</b>	<b>Random write, bulk incremental</b>
<b>Read pattern</b>	<b>Full table scan, partition table scan</b>	<b>Random read, small range scan, or table scan</b>
<b>Hive (SQL) performance</b>	<b>Very good</b>	<b>4-5x slower</b>
<b>Structured storage</b>	<b>Do-it-yourself / TSV / SequenceFile / Avro / ?</b>	<b>Sparse column-family data model</b>
<b>Max data size</b>	<b>30+ PB</b>	<b>~1PB</b>

# HBase vs. RDBMS

50

	<b>RDBMS</b>	<b>HBase</b>
<b>Data layout</b>	<b>Row-oriented</b>	<b>Column-family-oriented</b>
<b>Transactions</b>	<b>Multi-row ACID</b>	<b>Single row only</b>
<b>Query language</b>	<b>SQL</b>	<b>get/put/scan/etc *</b>
<b>Security</b>	<b>Authentication/Authorization</b>	<b>Work in progress</b>
<b>Indexes</b>	<b>On arbitrary columns</b>	<b>Row-key only</b>
<b>Max data size</b>	<b>TBs</b>	<b>~1PB</b>
<b>Read/write throughput limits</b>	<b>1 000s queries/second</b>	<b>Millions of queries/second</b>

# When to use HBase

51

## □ 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 many 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

# Case Study: Internet Memory

52

- At Internet Memory, they use HBase as a large-scale repository for the collections, holding terabytes of web documents in a distributed cluster.
- This article presents the data model of HBase, and explains how it stands between relational DBs and the "No Schema" approach.

# Understanding the HBase data model

53

- In 2006, the Google Labs team published a paper entitled BigTable: A Distributed Storage System for Structured Data.
- It describes a distributed index designed to manage very large datasets ("petabytes of data") in a cluster of data servers.
- BigTable supports key search, range search and high-throughput file scans, and also provides a flexible storage for structured data.
- HBase is an open-source clone of BigTable, and closely mimics its design.
- HBase is often assimilated to a large, distributed relational database.
- It actually presents many aspects common to "NoSQL" systems: distribution, fault tolerance, flexible modeling, absence of some features deemed essential in centralized DBMS (e.g., concurrency), etc.

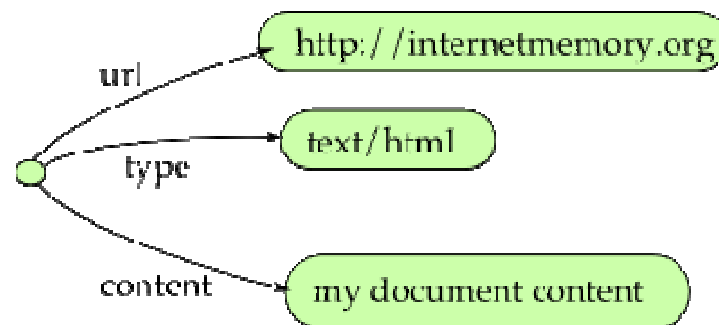
# The map structure: representing data with key/value pairs

54

- Key/Value pairs constitute a simple and convenient way of representing the properties of an object. Here is a running example the description of a Web document. For instance, using the JSON notation:

```
{  
  'url': 'http://internetmemory.org', type: 'text/html', content: 'my document content'  
}
```

- One obtains what is commonly called an associative array, a dictionary, or a map.
- Given a context (the object/document), the structure associates values to keys.
- We can represent such data as a graph, as shown by the figure below.
- The key information is captured by edges, whereas data values reside at leaves.



# There exists many possible representations for a map.

55

- We showed a JSON example above, but XML is of course an appropriate choice. At first glance, a map can also be represented by a table.
- The previous example is equivalently viewed as

url	type	content
http://internetmemory.org	text/html	my document content

- However, this often introduces some confusion. It is worth understanding several important differences that make a map much more flexible than the strict (relational) table structure.
  - ▣ there is no schema that constrains the list of keys (unlike relational table where the schema is fixed and uniform for all rows),
  - ▣ the value may itself be some complex structure.
- HBase, following BigTable, builds on this flexibility. First, we can add new key-value pair to describe an object, if needed.
- This doesn't require any pre-declaration at the *schema level*, and the new key remains local
- Other objects stored in the same HBase instance remain unaffected by the change.
- Second, a value can be another map, yielding a multi-map structure.

# An HBase "table" is a multi-map structure

56

- Instead of keeping one value for each property of an object, HBases allows the storage of several versions.
- Each version is identified by a timestamp. How can we represent such a multi-versions, key-value structure?
- HBase simply replaces atomic values by a map where the key is the timestamp.



# The extended representation for the example

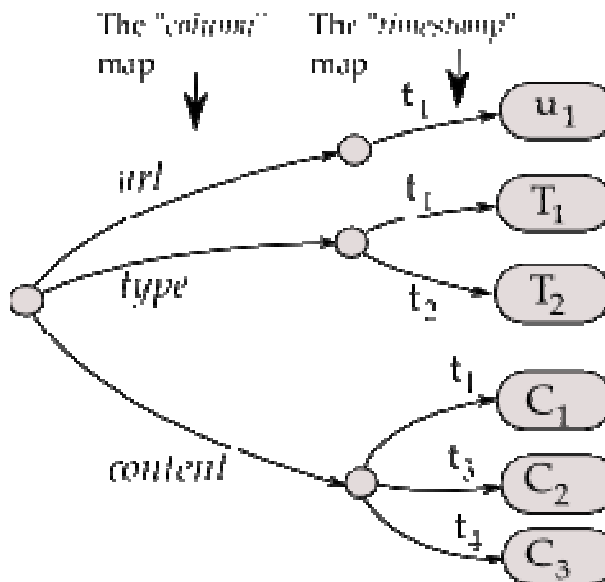
57

- It helps to figure out the power and flexibility of the data representation.
- Now, the document is built from two nested maps,
  - ▣ a first one, called "column" in HBase terminology (an unfortunate choice, since this is hardly related to the column relational concept),
  - ▣ a second "timestamp" (each map is named after its key).

# The document is globally viewed as a column map

58

- If we choose a column key, say, type, we obtain a value which is itself a second map featuring as many keys as there are timestamps for this specific column.
- In this example, there is only one timestamp for url (well, we can assume that the URL of the document does not change much).
- Looking at type and content, we find the former has two versions and the latter three.
- Moreover, they only have one timestamp ( $t_1$ ) in common.
- Actually, the "timestamp" maps are completely independent from one another.



# We can add as many timestamps

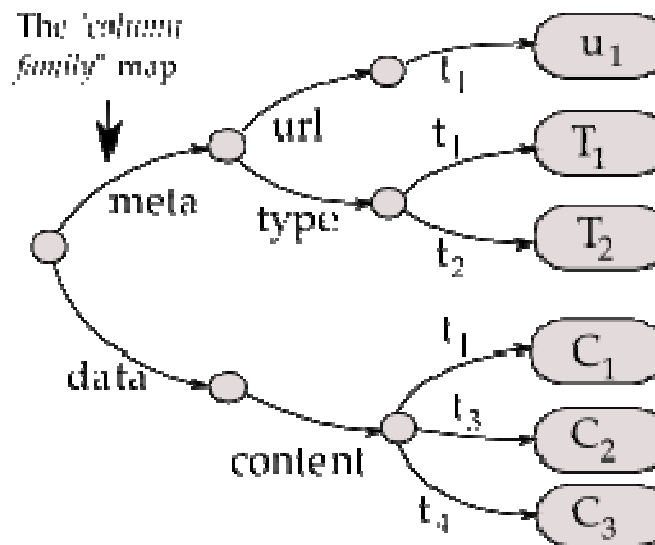
59

- Note that we can add as many timestamps (hence, as many keys in one of the second-level maps) as we wish.
- And, in fact, this is true for the first-level map as well: we can add as many columns as we wish, at the document level, without having to impose such a change to all other documents in a same HBase instance.
- In essence, each object is just a self-described piece of information (think again to the flexible representation of semi-structured data formats like XML or JSON).
- In this respect, HBase is in the wake of other 'NoSQL' systems, and its data model shares many aspects that characterize this trend: no schema and self-description of objects.
- Columns are grouped in column families, and a family is actually a key in a new map level, referring to a group of columns.

# Column Families

60

- In the Figure below, two families are defined: meta, grouping url and type, and data representing the content of a document.



- Unlike the column and timestamp maps, the keys of a family map are fixed.
- We cannot add new families to a table once it is created.
- The family level constitutes therefore the equivalent of a relational schema, although, as we saw, the content of a family value may be a quite complex structure.

# The full picture: rows and tables

61

- So, now, we know how to represent our objects with the HBase data model.
- It remains to describe how we can put many objects (potentially, millions or even billions of object) in HBase.
- This is where HBase borrows some terminology to relational databases: objects are called "rows", and rows are stored in a "table".
- Although one could find some superficial similarities, this comparison is a likely source of confusion.
- Let us try to list the differences:
  - ▣ 1.a "table" is actually a map where each row is a value, and the key is chosen by the table designer.
  - ▣ 2.we already explained that the structure of a "row" has little to do with the flat representation of relational row.
  - ▣ 3.regarding data manipulation, the nature of a "table" implies that two basic operations are available: put(key, row) and get(key): row. Nothing comparable to SQL here!

# Table map is a sorted map

62

- Finally, it is worth noting that the "table" map is a sorted map:
  - ▣ rows are grouped on the key value
  - ▣ 2 rows close to each other (with respect to the keys order) are stored in the same area.
- This makes possible (and efficient) range queries of keys.
- The following Figures summarize the structure for an hypothetical webdoc HBase table storing a large collection of web documents.
- Each document is indexed by its url (which is therefore the key of the highest level map).
- A row is itself a local map featuring a fixed number of keys defined by the family names (f1, f2, etc.), associated to values which are themselves maps indexed by columns.

