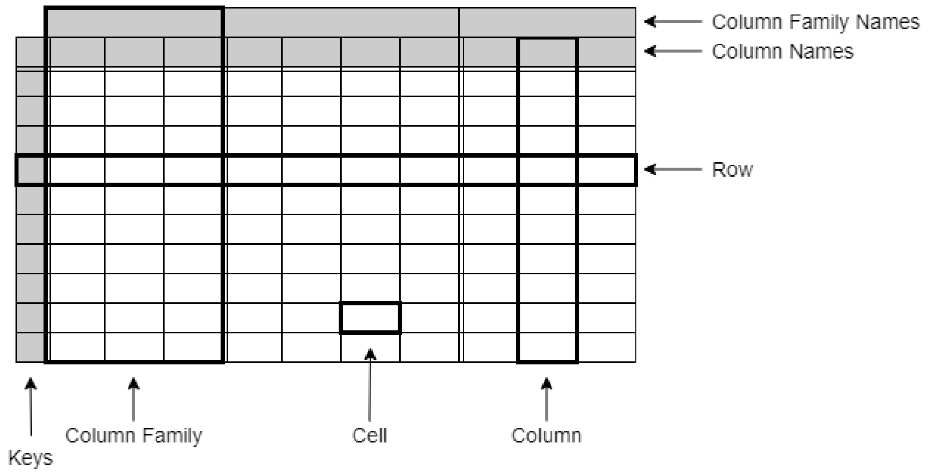
CS4650 Topic 10:
HBase

HBase

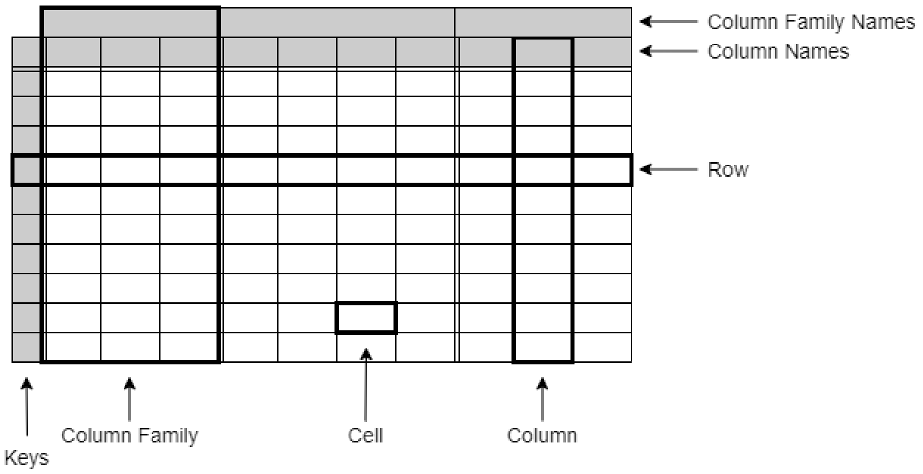
- HBase is a distributed column-oriented database built on top of HDFS.
- HBase provides real-time read/write random access to very large datasets.
- While HBase has some similarities with a relational database, there are significant differences:
    - There are no relationships between multiple tables
    - The SQL language is not supported
    - Joins and complex queries are not available
    - It can be distributed
    - Scalability is built-in

HBase Diagram



- HBase contains a number of named tables.
- Tables are made of rows and columns. At the intersection of a row and a column is a cell.
- Each row is labeled by a unique key, corresponding to the primary key of a relational database.
- Each column is labeled by a column name.

HBase Diagram



- Columns are grouped into sets called column families.
- Each column family is labeled with a name.
- When the table is constructed, all of the column families are defined for the table.
- Columns can be inserted at any time. Each column must belong to one of the column families.
- HBase tables are usually very sparse. There might be a million columns in the table, but any row may only have a small number of cells present.

HBase Cell Details

- A cell's contents is an uninterpreted array of bytes. This means that HBase doesn't care what data is stored, any values can be placed there. The values are only inserted or retrieved, they are never interpreted.
- The contents are *versioned*. When a value is inserted, the timestamp of when the insertion happened is used as the version number.
- As the table is being configured, the number of versions can be specified. Each cell can contain up to this number of versions.

Regions

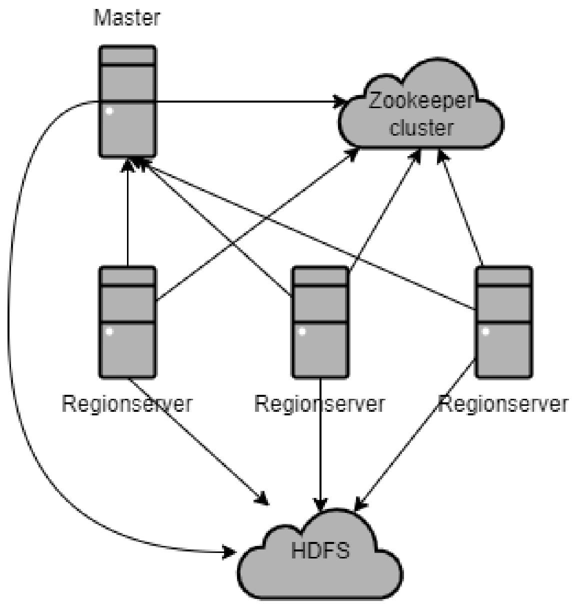
- Tables are automatically partitioned by HBase into *regions*.
- Each region consists of a series of rows, from a *first row* (inclusive) to a *last row* (exclusive).
- Initially, the table has one region containing all of the rows.
- As rows are added to the region, eventually a threshold is crossed, at which point the region is split into two of approximately equal size.
- Regions are distributed over the HBase cluster, so each node only hosts a subset of the table's total regions.

Regions

- If there are many columns and column families in a table, the table may also be split by columns.
- However, a column family is always kept together, all of the columns within that family are in the same HDFS file.

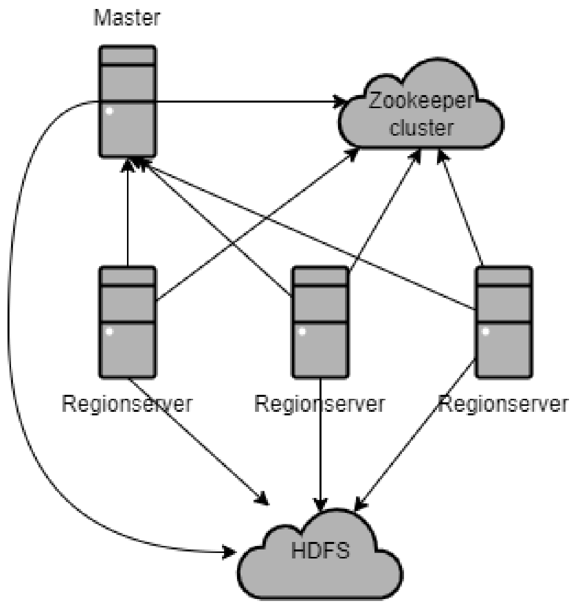
Architecture

- The HBase files are stored on HDFS.
- One server is the HBase Master which oversees the operation of HBase.
- The actual HBase files are managed by Regionservers.
- Each Regionserver may hold one or more Region of an HBase table.
- ZooKeeper is a distributed NoSQL database holding graphical information, the 'file structure' of the HBase database.



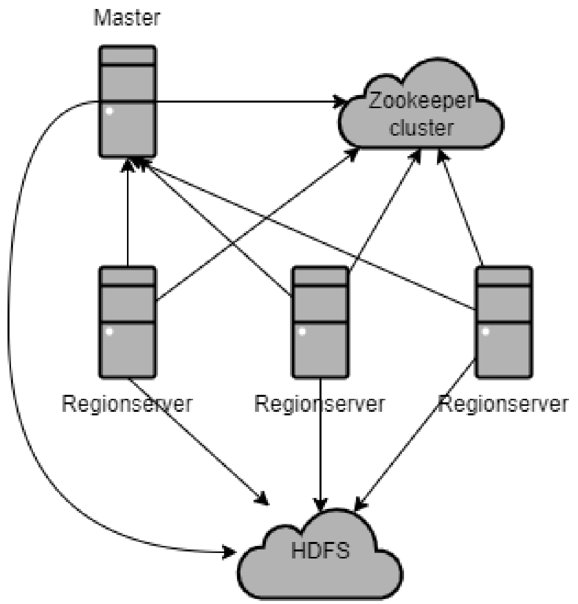
Architecture

- The HBase master is responsible for:
  - Bootstrapping a new installation,
  - Assigning regions to registered Regionservers,
  - Recovering from Regionserver failures.
- The master node is lightly loaded.



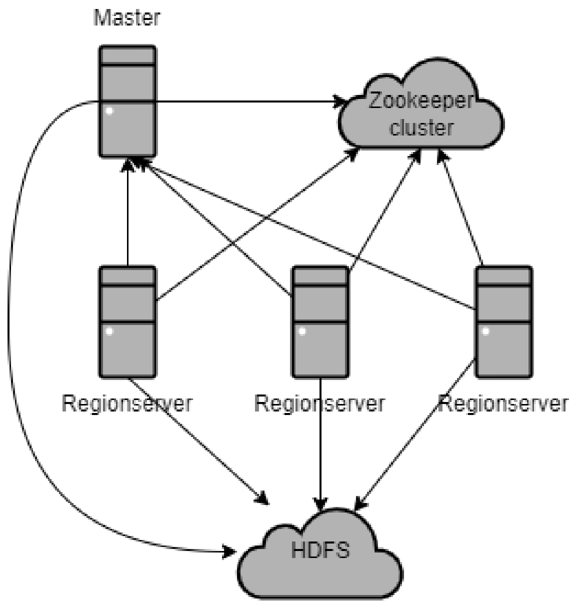
Architecture

- Regionservers carry zero or more regions.
- Regionservers field client read/write requests.
- A Regionserver also manages Region splits, informing the HBase master about the new daughter Region.



Architecture

- The HBase Master manages a ZooKeeper instance, which is the authority on cluster state.
- ZooKeeper holds the root catalog table and address of the current Master.
- Assignment of Regions is mediated via ZooKeeper in case servers crash mid-assignment.
- Once initialized, the Regionservers keep a handle to ZooKeeper.



Architecture

- HBase keeps file system structure information in two special catalog tables, -*ROOT*- and *.META.*, within which it maintains the current list, state, and location of all regions in the cluster.
- *-ROOT-* holds a list of all the *.META.* table regions.
- *.META.* holds the list of all the Regions.
- Entries in these tables are keyed by the region name (which is the table name, the region's start row, and creation timestamp).
- As Regions transition -- are split, disabled/enabled, deleted, or redeployed -- these catalog tables are updated.

Writing Data

- When a Regionserver receives a write request, the information is first written to a commit log.
- The results are then added to an in-memory *memstore*.
- When the memstore fills, its contents are flushed to the filesystem.
- If the regionserver crashes:
    - The Master discovers that the regionserver has crashed.
    - The regionserver's commit log is split by region.
    - Regions that were on the dead regionserver replay their portion of the commit log, so they are brought up-to-date before being open for business.

Reading Data

- When a read request arrives at a region server, the memstore is consulted first (the in-core version). If the memstore has enough information to complete the request, the query is completed there.
- If additional data is required, flush files are consulted in order, from newest to oldest, until either the query is satisfied or until there are no more flush files (in which case, there is no more data to add to the query).

Compacting Data

- Once the number of flush files reaches a given threshold, a background process compacts flush files, combining several files into one. This increases the performance of the system.
- During compaction, the system clears out:
    - Old versions of cells, once the schema's maximum version count is exceeded.
    - Deleted cells,
    - Expired cells.
- A separate process monitors the flush file sizes, splitting the region when they grow in excess of the configured maximum.

Installing HBase

- A stable release of HBase can be downloaded from:
  http://www.apache.org/dyn/closer.cgi/hbase/
- Unback the zip file.
- Configure HBase to find your Java installation. If you have set the JAVA_HOME environment variable, HBase can find it.
- Add the HBase binary directory to your command-line path.

Test Drive

- To start a temporary instance of HBase using the */tmp* directory of your local file system, in a shell type:

```
% start-hbase.sh
```

- Next start the HBase shell by typing

```
% hbase shell
```

- Use the *help* command to find a list of all commands, or *help Group* to find help on a particular category, or *help Command* to find help on a specific command.

Test Drive

- We will create a table.
- To do this, we give the name of the table, and define its schema (which means we list the column families of the table). Recall that the column families must be defined once and for all in this creation step. Conversely, new columns can be added to existing families at any time.
- When a column family is defined, you can also indicate whether the data in this family should be compressed, and you can state the number of versions that should be kept of any cell in this column family.
- The following command creates a table, *test*, and one column family, *data*:

```
> create 'test', 'data'
```

Test Drive

- To verify that the table was correctly built, run the following:

```
> list
test
```

Test Drive

- Recall that a column name is prefixed with the column family name.
- We have created a column family named 'data'.
- To reference a column within that family, give the family name, a colon, then the name for the column. For example, we can reference the '1' column using 'data:1'.
- If the indicated column exists, it is used.
- If that column does not exist, it will be created (on a put), or return null (on a get).

Test Drive

- We can add three rows of data to the database:

```
> put 'test', 'row1', 'data:1', 'value1'
…
> put 'test', 'row2', 'data:2', 'value2'
…
> put 'test', 'row3', 'data:3', 'value3'
…
> scan 'test'
ROW      COLUMN+CELL
 row1    column=data:1, timestamp=..., value=value1
 row2    column=data:2, timestamp=..., value=value2
 row3    column=data:3, timestamp=..., value=value3
```

Test Drive

- The database can be deleted as follows:

```
> disable 'test'
…
> drop 'test'
…
> list
0 rows(s)...
```

- Shut down HBase instance by running:

```
% stop-hbase.sh
```

Test Drive in Java

- For the next several slides, we will see how to do this 'test drive' using a Java interface.
- The examples do not show the package names nor imports.
- All of the examples below 'run together' (values from earlier slides may be used on later slides).

Getting Started

```
public class ExampleClient
{
        public static void main(String[] args) throws IOException
        {
                Configuration config = HBaseConfiguration.create();

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

- The Configuration reads the HBase configuration, and is then used to create an *admin* and a *table*.
- The Admin is used to administer the cluster, in this case to add and drop tables.

Creating Table With One Column Family

```
                HTableDescriptor htd = new HTableDescriptor("test");
                HColumnDescriptor hcd = new HColumnDescriptor("data");
                htd.addFamily(hcd);
                admin.createTable(htd);
        byte [] tablename = htd.getName();
```

- We build the description of the table, giving its name, then listing each of the column families to be used.
- In this case, we only have one column family.
- We can also set the version count and compression flags, if desired.
- The *admin* is then used to create the table.
- We also grab the table name here, to be used in subsequent commands.

Doing a Put

```
                    HTable table = new HTable(config, tablename);
                    byte [] row1 = Bytes.toBytes("row1");
                    Put p1 = new Put(row1);
                    byte [] databytes = Bytes.toBytes("data");
                    p1.add(databytes, Bytes.toBytes("1"),
Bytes.toBytes("value1"));
                    table.put(p1);
```

- We fetch a handle to the table, asking the configuration to find the table with the given name.
- We next create a Put (with the id of "row1").
- The *add* command takes the column family, column name, and value for the cell. We can do multiple adds if desired.
- We then 'put' the data to the table.

Doing a Get

```
Get g = new Get(row1);
Result result = table.get(g);
System.out.println("Get: " + result);
```

- Here we build a *Get*, passing the ID of the desired row.
- The table then performed the Get, returning a result.
- In this example, we simply print the complete result.

Doing a Scan

```
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
try
{
        for (Result scannerResult: scanner)
        {
                System.out.println("Scan: " +
scannerResult);
        }
}
finally
{
        scanner.close();
}
```

- This scans the table, printing all of the contents.

Drop the Table

```
                    admin.disableTable(tablename);
                    admin.deleteTable(tablename);
          }
}
```

- Finally we delete the table.
- Note that we have to disable a table before we delete it.

See The Book

- The book has additional examples that use HBase:
  - One example shows how to run a Map/Reduce job on the HBase.
  - Another example shows how to build an HBase table to hold the weather data from some of our previous examples. This example describes the schemas for the tables, and shows a Map/Reduce program that can load the data into the HBase database.
  - A third example shows how to build a web interface that allows users to access the weather data, showing how to retrieve station information and how to retrieve ranges of rows of weather data.

HBase vs RDBMS

- Hadoop and HDFS does not allow random reads and writes of data, so does not compare favorably with an RDBMS for applications where the data can change.
- For very large data sets, with a very large number of rows and a very large number of columns, an RDBMS cannot handle these sizes.
- If strong consistency, referential integrity, and complex queries are needed (the majority of small- to medium-volume applications), there is no substitute for the ease of use, flexibility, maturity and power of an RDBMS solution.
- For a scalable database with read/write concurrency, HBase is the preferred solution.