HBase

1. Introduction

HBase is a distributed column-oriented database built on top of HDFS. It provides real-time read/write random-access to very large datasets.

Data Model

Applications store data into labeled tables, which are made up of rows and columns. Table cells are **versioned**. By default, their version is a time-stamp auto-assigned by HBase at the time of cell insertion. A cell's content is an uninterpreted array of bytes. **Table row keys** are byte arrays, hence any data structure can serve as the row keys when serialized into binary data. Table rows are sorted by table row key, which are the table's primary key. All table accesses are via the primary key. Row columns are grouped into **column families**, and all column family members have a common prefix. For example, the columns *temperature:air* and *temperature:dew_point* are both members of the temperature family, and *station:identifier* belongs to the *station* family. A column family prefix must be of printable characters, the qualifying tail can be made of any arbitrary bytes.
A table's column families must be specified up front as part of the table schema definition, but new column families can be added on demand. For example a new column *station:address* can be added provided the column family *station* already exists on the target table.
Physically, all column family members are stored together on the filesystem. Tunings and storage specifications are done at the column family level.
Tables are automatically partitioned horizontally by HBase into **regions**. Each region comprises a subset of a table's rows. A region is defined by its first row, inclusive, and last row, exclusive, plus a randomly generated region identifier. Initially when the table is small, it comprises a single region, on a single host. When the table grows, it is split into various regions of approximately equal size and distributed over an HBase cluster.

Implementation

A HBase is characterized with a HBase **master** node orchestrating a cluster of one or more **regionserver** slaves. The *master* is responsible bootstrapping a virgin install, assigning regions to *regionservers* and recovering *regionserver* failures. The *regionservers* carry zero or more regions and field client read/write requests. They manage region splits informing the *master* about new daughter regions. HBase depends on **ZooKeeper**.
Regionserver slave nodes are listed in the *conf/regionservers* file. Cluster site-specific configuration is made in the *conf/hbase-site.xml* and *conf/hbase-env.sh* files

Internally, HBase keeps special catalog tables named **-ROOT-** and **.META.** within which it maintains the current list, state, recent history and location of all regions on the cluster. The -ROOT- table holds the list of .META. table regions. The .META. table holds the list of all user-space regions. Entries in these tables are keyed using the region's start row. As row keys are sorted, finding the region that hosts a particular row is a matter of looking the first entry whose key is greater than or equal to the requested row key. The catalog tables are updated as regions are edited.

Clients connect to the *ZooKeeper* cluster to learn the location of -ROOT-, clients will consult the -ROOT- to elicit the location of the .META. region which scope covers that of the requested row. The client then does a lookup against the found .META. region to locate the row. Thereafter the client interacts directly with the hosting *regionserver*. Clients cache all they learn traversing -ROOT- and .META. including locations and user-space region start and stop rows so they can figure hosting regions without repeatedly consulting -ROOT- and .META. table. Clients will continue using this cache until a fault occurs, and when it happens it will consult .META. or -ROOT-.

Writes arriving at a regionserver are first appended to a commit log and then added to an in-memory cache. When this cache fills, its content is flushed to the filesystem. This commit log is hosted on HDFS.

When reading, the region's memcache is consulted first. If sufficient versions are found to satisfy the query, it is returned. Otherwise flush files are consulted in order from newest to oldest till sufficient versions are found or till we run out of flush files to consult.

2. Installation and First Run

Installing HBase

Download a stable release from the HBase release page (http://apache.oss.eznetsols.org/hbase/) and unpack it onto filesystem:

```
% tar xzf hbase-m.n.o.tar.gz
```

Set JAVA_HOME environmental variable to point to a Java installation. You can also edit Hbase's *conf/hbase-env.sh* and specifying the JAVA_HOME variable. Add HBase binary directory to command-line path:

```
% export HBASE_INSTALL=/home/hbase/hbase-m.n.o
% export PATH=$PATH:$HBASE_INSTALL/bin
```

Test if installation is successful by typing:

```
% hbase
```

The usage instructions will be displayed.

First Run

Start a temporary instance of HBase that uses the */tmp* directory on the local filesystem for persistence:

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

This will launch two daemons: a standalone HBase instance that persists to the local filesystem (by default to */tmp/hbase-${USERID}*) and a single instance of ZooKeeper to host cluster state. To administer your HBase instance, launch the HBase shell by typing:

```
% hbase shell
… some usage information and version information …
hbase(main):001:0>
```

This will bring up a JRuby IRB interpreter that has some HBase-specific commands added. Type help and hit RETURN to see the list of shell commands and usage instructions.

Simple Example

Create a simple data, add some data, and clean up. To create a table, you must name the table and define its **schema**. A table's schema comprises table **attributes** and the list of table column families. Column families themselves have attributes which you set at schema definition time. Examples of column families attributes include how many versions of a cell to keep. Schemas can be edited later by taking the table offline using the `disable` command, editing the table using `alter`, and putting the table back online using `enable`. Create a table named `test` with a single column family named `data` using defaults for table and column family attributes:

```
hbase(main):007:0> create ‘test’, ‘data’
0 row(s) in 4.234seconds
```

See the `help` output for examples adding table and column family attributes when specifying a schema. Run the `list` command to output all tables in the user space:

```
hbase(main):019:0> list
test
1 row(s) in 0.148 seconds
```

Insert data into three different rows and columns in the data column family, and then list the table content:

```
hbase(main):021:0> put ‘test’, ‘row1’, ‘data:1’, ‘value1’
0 row(s) in 0.0431 seconds
hbase(main):022:0> put ‘test’, ‘row2’, ‘data:2’, ‘value2’
0 row(s) in 0.0332 seconds
hbase(main):023:0> put ‘test’, ‘row3’, ‘data:3’, ‘value3’
0 row(s) in 0.0404 seconds
hbase(main):024:0> scan ‘test’
ROW         COLUMN+CELL
 row1       column=data:1, timestamp=1240148026198, value=value1
 row2       column=data:2, timestamp=1240148040035, value=value2
 row3       column=data:3, timestamp=1240148047497, value=value3
3 row(s) in 0.083 seconds
```

To remove the table, first disable it, then drop it:

```
hbase(main):025:0> disable ‘test’
27/3/12 16:01:10 INFO client.HBaseAdmin: Disabled test
0 row(s) in 6.0234 seconds
hbase(main):026:0> drop ‘test’
```

```
27/3/12 16:01:15 INFO client.HBaseAdmin: Deleted test
0 row(s) in 0.0234 seconds
hbase(main):027:0> list
0 row(s) in 2.0642 seconds
```

Shut down HBase instance:

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

3. HBase on distributed HDFS

First, make sure the you have HDFS up and running, you only need to run start-dfs.sh, HBase does not require MapRed to be running. Set up Zookeeper in the replicated mode.
Unpack HBase to a directory. The important files to edit are inside `conf` folder.

```
	<!-- conf/hbase-site.xml -- >
	<configuration>
	<property>
		<name>hbase.master</name>
		<value>master:60000</name>
	</property>
	<property>
		<name>hbase.cluster.distributed</name>
		<value>true</name>
	</property>
	<property>
		<name>hbase.rootdir</name>
		<value>hdfs://master:54310/hbase</value>
		<description>refer to hadoop configuration for the domain and
		port you used for dfs.namenode.dir</description>
	</property>
	<property>
		<name>hbase.zookeeper.property.clientPort</name>
		<value>2181</name><!-- this is default value -->
	</property>
	<property>
		<name>hbase.zookeeper.quorum</name>
		<value>master,slave</name>
		<description>list all the servers in the quorum here</description
	</property>
	<property>
		<name>hbase.zookeeper.property.dataDir</name>
		<value>/home/hduser/zookeeper/tmp</name>
		<description>folder for zookeeper to store snapshots</
description>
	</property>
	</configuration
	<!-- in hbase-env.sh -- >
	export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
```

```
        <!-- set to the location of your java -- >
        …
        export HBASE_MANAGES_ZK=false
        <!-- default is true, use default if testing HBase on a locahost -- >

        <!-- in regionservers -- >
        master
        slave
        <!-- list all RegionServers, one per line -- >
```

Also it will be convenient to edit your $HOME/.bashrc file to include these lines:

```
        export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
        export HADOOP_HOME=/home/hduser/hadoop
        export ZK_HOME=/home/hduser/zookeeper
        export HBASE_HOME=/home/hduser/hbase
        export PATH=$PATH:$HADOOP_HOME/bin:$ZK_HOME/bin:$HBASE_HOME/bin
```

Start Hadoop hdfs:

```
        $ start-dfs.sh
```

Start ZooKeeper on all machines in quorum:

```
        $ zkServer.sh start
```

Start HBase:

```
        $ start-hbase.sh
```

A web interface is up at

```
        master:60010
```

Scroll to the bottom to the heading Region Servers, if you see all the Region Servers that you have configured listed there, set-up was successful.

Troubleshooting

Problems encountered include:
1) on the logs of remote slave RegionServers, it is observed that slave tries to connect to localhost:60000, when it it suppose to connect to master:60000. Tracking this to the log files on master, observed that master identifies itself as localhost:60000 (perhaps because I set it to be a regionserver as well), and passes this information directly to remote slave. Problem was solved by either of two ways: 1. Commenting out the line the assigns 127.0.0.1 to the current machine name in /etc/hosts, and 2. Commenting out all IPv6 lines in /etc/hosts (this might not have anything to do with the problem, have not yet tested).

4. HBase and MapReduce

HBase classes and utilites in the org.apache.hadoop.hbase.mapred package facilitate using HBase as a source and/or sink in MapReduce jobs. The TableInputFormat class makes splits on region boundaries so maps are handed a single region to work on. The TableOutputFormat will write the result of reduce into HBase. Refer to the `RowCounter` code for an example of running a map task to count rows using TableInputFormat.

We can copy raw input data onto HDFS and then run a MapReduce job that can read the input and write to HBase. Refer to `HBaseTemperatureImporter` for an example of a MapReduce job that imports observations from a raw tab delimited file using TextInputFormat to HBase.

To implement web application, we will use the HBase Java API directly. An example of a simple query is to get static station information, seen in `getStationInfo`.
We can also use HBase scanners for retrieval of observations in our web application, seen in `getStationObservations`.

5. HBase and Pig

HBase implementation of Pig via HBaseStorage class in org.apache.pig.backend.hadoop.hbase. HBaseStorage is an implementation of LoadFunc and StoreFunc in pig. An example of how to load data from HBase:

```
grunt> raw = LOAD ‘hbase://SampleTable’
>>    USING org.apache.pig.backend.hadoop.hbase.HBaseStorage(
>>    ‘info:first_name info:last_name friends:* info:*’,
>>    ‘-loadKey true -limit 5’)
>>    AS (id:bytearray, first_name:chararray, last_name:chararray,
>>    friends_map:map[], info_map:map[]);
```

Row key is inserted first in the result schema if `-loadKey` is set to `true` (`id:bytearray` is the rowkey from SampleTable). This example loads data redundantly from `info:` column family to show usage. To load all columns from column family info, specify as info or `info:*`. To fetch columns in the column family info that starts with `bar`, specify `info:bar*`. The resulting tuple will always be the size of the number of tokens specified in the column list. Items in a tuple will be scalar values when a column is specified, or a map of column descriptors to values when a column family is specified. To store data into HBase:

```
grunt> copy = STORE raw INTO ‘hbase://SampleTableCopy’
>>    USING org.apache.pig.backend.hadoop.hbase.HBaseStorage(
>>    ‘info:first_name info:last_name friends:* info:*’)
>>    AS (info:first_name info:last_name buddies:* info:*);
```

`STORE` will expect the first value in the tuple to be the row key (in this case `id:bytearray` becomes the row key for `SampleTableCopy`). Scalar values need to map to an explicit column descriptor while maps need to map to a column family name. Specify the `AS` clause will allow you to rename the column or column family. Above, the friends column family data from `SampleTable` will be written to a buddies column family in the `SampleTableCopy` table.