INTRODUCTION TO HBASE

Overview

Hbase is the open source implementation of Google's Big Table database, which is where Google stores data for, for example, Google Earth and web index data.

HBase is a structured noSQL database that rides atop Hadoop. You can store Hbase data in the HDFS (Hadoop Distributed File System). And you can also store HBase data in Amazon S3, which has an entirely different architecture. HBase is structured because it has the row-and-column structure of an RDBMS, like Oracle. But it a column-oriented database and not a row-oriented one, which we explain below. And it is noSQL because you cannot use SQL to build relations between tables as you can with a RDBMS, like JOIN.

What HBase does is provide random access to big data. Hadoop does not: it is a batch system that only writes files but does not update them.

Like Cassandra, HBase stores data in a memory table and then flushes it to storage in massive writes to disk. So that gives it the high throughput needed for big data applications.

Architecture

HBase has a master-slave architecture. The **master server** coordinates **region servers**. The region servers which are responsible for writing, reading, and other data operations.

Apache Zookeeper, which is installed with Hadoop, handles configuration and distributed operations

Install HBase

Below we give some examples of how to create tables in HBase and write data. So you need to install HBase first. You can find instructions for installing HBase on the internet. There is no RPM package for it. Basically you have to install Hadoop first. Then you download HBase and unzip it and make small changes to the Hadoop configuration file hbase-site.xml. So its installation is not complicated.

We will use HBase in standalone instead of distributed mode for these examples.

HBase schema

HBase schemas will be strange-looking for the reader who is familiar with Oracle or MySQL. First let's give some definitions to help clarify that. We will use a product database for our examples. But that does not mean HBase is suitable for an inventory control system. It could be, especially for one that is extremely large.

- **Table**—a collection of rows.
- **Row**—a collection of column families.
- **Column Family**—a collection of columns. HBase stores data by column family and not row. This makes for faster retrieval of columns since they are located near each other.
- **Column**—are individual data items, like product price. For example, in a system designed to store product information you could have a column family called characteristics and then another called inventory. Characteristics could contain the columns description, manufacturer, and serial number. Inventory could include itemCount, SKU (stock keeping unit), EAN (barcode Europe), and UPC (barcode USA). You reference the columns like this characteristics:itemCount.
- **Timestamp**—HBase indexes row keys, columns, and timestamp. The timestamp can be any time format including simple integers which are not time at all. Because timestamp is part of the key, you can have duplicate rows. That means you can have multiple versions of a row. For example you could have an accounting transaction at time n and the same information at some other time.
- **Row Key**—The row key is whatever you store in the row key column. So it's not just an ordinal number. In this example, we use product number. You do not give the row key a name, like productNumber, like you do with columns. HBase keeps row keys in alphabetical order so that similar items are kept next to each other. For example, in the Google documentation that explains Google Big Table they use the example of their web index. Data for abc.com is kept next to sales.abc.com. (In other to store those next to each other Google stores those domain names backwards like com.abc and com.abc.sales.)
- **Cell**—Technically HBase stores data in maps. Python, Scala, and Java programmers know that a map is a (key->value) data structure, with no duplicate keys allowed. Google says that HBase is a "sparse, consistent, distributed, multidimensional, sorted map." Data is stored as this map ((rowkey, column family, column, timestamp) -> value). So you can say that a cell is a column value.That's not the exact technical definition but an easy way to think about it.

To illustrate, rows have the structure shown below with the row key, timestamp, column family, and columns. But not all rows need to have the same columns, which is a big change too from RDBMS. That is where the definition **sparse** comes in: rows without a specific column do not consume empty

space, like an RDBMS does, to store that empty value.

Row Key	TimeStamp	Column Family		Column Family	
		Column 1	Column 2	Column 3	Column 4

Suppose we have a product database. Products are stored in SKU (stock keeping unit) which can be a single item, a 6 pack, a case, etc. The UPC code is the bar code. And quantity is how much is on hand in inventory.

So we have this schema below with product number as the Row Key and two product families: Characteristics and Inventory. Each of those has columns so we have this sample data:

Product Number	TimeStamp	Characteristics			Inventory	
		Description	UPC	Manu.	sku	quantity
1	10	skateboard	123		each	100
2	10	glasses	456		package	400
2	15	glasses	456		package	600
3	20	boar	789	Mako	each	2

Notice that we have 2 rows with row key 2 and UPC column 456. The difference is the timestamp. So we have inventory levels at two different times.

Notice that product 3 has characteristics:manufacturer column and the other rows do not. This illustrates that rows have can have different columns. We will work with this schema in our examples below.

HBase shell

Put /usr/local/hbase/hbase(version)/bin/ into your path so that you can open the shell like this:

```
hbase shell
```

This is the JRuby IRB shell riding atop HBase. So you can create variables and so forth.

Create table using the shell

Create a table using:

```
create <table name>, <column family 1>,<column family 2>, ...
```

As in:

```
create 'products', 'characteristics’, 'inventory'
```

Type **list** to show that it is created and then **describe** command to show the schema.

Add data to table

Now, we do not create column names before you put data into the table. You do that on the fly. The command is:

```
put <table name>, <row key>, <column family: column>
```

That puts a value at a specific row and column. You cannot write to more than one column in one put.

```
put 'products', '123', 'characteristics:description', 'skateboard'
```

Write to another cell for the same product:

```
put 'products', '123', 'inventory:count', '100'
```

Then list data using scan:

```
scan 'products'

ROW                     COLUMN+CELL
 123                    column=characteristics:description,
timestamp=1488291808832, value=skateboard
 123                    column=inventory:count, timestamp=1488292001662,
value=100
```

Retrieve data

Here is how to retrieve a specific row/cell:

```
get 'products', '123', 'characteristics:description'

COLUMN                  CELL
characteristics:descrip timestamp=1488291808832, value=skateboard
```

Using versions

In order to use the timestamp feature to store multiple versions we create the table like this:

```
create 'products', {NAME => 'characteristics', VERSIONS => 3}, {NAME =>
'inventory', VERSIONS => 3}
```

That means keep up to 3 versions of each row/column.

Then write a value and do not give a timestamp. It will use the system time.

```
put 'products', '123', 'characteristics:description', 'skateboard'
```

Then write same row, column value and use any any integer for a timestamp. Here we use 2.

```
put 'products', '123', 'characteristics:description', 'skateboard', 2
```

Now scan to show the data in the table and you see we have two versions:

```
scan 'products', {VERSIONS => 3}

ROW                     COLUMN+CELL
 123                    column=characteristics:description,
timestamp=1488293615489, value=ska
```

```
                        teboard
 123                    column=characteristics:description, timestamp=2,
value=skateboard
```

Delete a table

To delete a table you must first disable it:

```
disable ‘products’
```

Then you can delete it:

```
drop 'products'
```

Note that we put quote marks around the name table name.

Data types

Notice that we have been using strings for all our examples. Hbase writes values as bytes so any object that can be stored as bytes (and that is everything but bits) can be stored. So it supports strings, numbers, images, and complex objects. You can see this in Java when you use:

```
Put(Bytes.toBytes());
```

Create a table using Java

Most programmers working with HBase use Java. HBase uses JRuby for a shell, which also is an interface to Java.

Here is an example of how to create a table using Java. You need to set the CLASSPATH as shown below in order to run this example. Basically the **hbase classpath** command returns a very long string that you need to compile your program.

```
export CLASSPATH=$CLASSPATH:`hbase classpath`
cat HbaseExamples.java:

import java.io.IOException;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.conf.Configuration;

public class HbaseExamples {
        public static void main(String[] args) throws IOException {
                Configuration conf = HBaseConfiguration.create();
                HBaseAdmin admin = new HBaseAdmin(conf);
                HTableDescriptor tableDescriptor = new
                HTableDescriptor(TableName.valueOf("products"));
```

```
                tableDescriptor.addFamily(new
HColumnDescriptor("characteristics"));
                tableDescriptor.addFamily(new
HColumnDescriptor("inventory"));
                admin.createTable(tableDescriptor);
                System.out.println(" Table created ");
        }
}
```

Set CLASSPATH then compile and run the program:

```
export CLASSPATH=`hbase classpath`

javac -cp `hbase classpath` HbaseExamples.java
java -cp $CLASSPATH:. HbaseExamples
```

It should echo lots of Hbase messages to the console ending with:

```
2017-02-27 16:06:01,616 INFO   client.HBaseAdmin: Created products
 Table created
```

Now you can open the shell and list and see that products table has been created:

```
hbase(main):009:0> list
TABLE
products
```

So that is enough information to get you started. Before you dive in and start using HBase you should study further how to create masters and regions and schemas so that you build your data in a way that is consistent with its architecture. That means getting rid of the RDBMS bias in your thinking, which those of us who have MySQL and Oracle for years will have. And you could study some JRuby to see how you could use variables in your HBase scripts.