HandOut.7: Delving Deep into Hbase

Installation

Access your docker working directory

Execute: git clone https://github.com/big-data-europe/docker-hbase.git

Execute (Hbase in standalone mode): docker-compose -f docker-compose-standalone.yml up -d

Check all containers: docker ps

Check your IP: ipconfig /all

[ToDo: Paste relevant snapshot at port 16010]

Execute: docker exec -it hbase /bin/bash

Execute: hbase shell

General Shell Commands

Starting HBase Shell: hbase shell

Exiting HBase Shell: exit

View existing tables in HBase: list

View existing servers in HBase: status

View version of HBase in use: version

To open guide for tables in HBase: table_help

Create table in HBase:

create'ecommerce_transactions,'amount','card_type', 'websitename', 'countryname', 'datetime', 'transactionID', 'cityname', 'productname'

Insert some rows for tables in HBase:

put 'ecommerce_transactions', '2', 'card_type', 'MasterCard'

put 'ecommerce_transactions', '3', 'card_type', 'Visa'

put 'ecommerce_transactions', '4', 'card_type', 'MasterCard'

put 'ecommerce_transactions', '5', 'card_type', 'Maestro'

put 'ecommerce_transactions', '1', 'amount', '50.87'

put 'ecommerce_transactions', '2', 'amount', '1023.2'

put 'ecommerce_transactions', '3', 'amount', '3321.1'

put 'ecommerce_transactions', '4', 'amount', '234.11'

put 'ecommerce_transactions', '5', 'amount', '321.11'

Get transactions for only row value 2 and 3 in HBase:

get 'ecommerce_transactions', '2'

get 'ecommerce_transactions', '5'

Get description of the table in HBase: describe 'ecommerce_transactions'

Alter is the command used to make changes to an existing table. This command can help change the maximum number of cells of a column family, set and delete table scope operators, and delete a column family from a table:

alter 'ecommerce_transactions', NAME => ‘card_type’, VERSIONS =>5

Deleting a cell in a table in HBase: delete 'ecommerce_transactions', ‘4’, ‘card_type’

Deleting a row in a table in HBase: deleteall 'ecommerce_transactions', ‘4’

Count number of rows in a table in HBase: count 'ecommerce_transactions'

Disable, drop and recreate a table in HBase: truncate 'ecommerce_transactions'

Check if a table is present in HBase: exists 'ecommerce_transactions'

View all rows in a table of HBase: scan ‘ecommerce_transactions’

Disable a table of HBase: disable ‘ecommerce_transactions’

Drop a table of HBase: drop ‘ecommerce_transactions’

Note: drop won’t work till table has been disabled

Scan Filter Exercises

[ToDo: Paste snapshot of 2 filter outputs]

Using the data inserted in 'ecommerce_transactions' table, enhancing reads of data present in HBase using the scan filtering techniques. There are 6 most commonly used filters, but these can also be combined together for more specific searches.

Check for available filters in HBase to read data more specifically: show_filters (enter in HBase shell)

Using the data set we put earlier perform the following hands-on exercise:

Using KeyOnlyFilter get the names of columns and column family qualifiers without seeing their values (typically used for a dataset with very large values and we just want to see the attributes):

```
scan 'ecommerce_transactions', {FILTER => "KeyOnlyFilter()"}
```

Using the PrefixFilter to find any pattern associated with our row keys: Check which row key represents which website for transactions?

```
scan 'ecommerce_transactions', {FILTER => "PrefixFilter('1')"}
```

Combining filters: Suppose now we want to see which column families are on row key 2? Then we combine the two filters such that:

```
scan 'ecommerce_transactions',{FILTER => "PrefixFilter('2') AND KeyOnlyFilter()"}
```

Using the ColumnPrefixFilter, find the column families which start with 'c'

```
scan 'ecommerce_transactions', {FILTER => "ColumnPrefixFilter('c')"}
```

Similarly, specify two arguments in this filter and find all column families which start with 'c' or 'p'

```
scan 'ecommerce_transactions', {FILTER => "MultipleColumnPrefixFilter('c','p')"}
```

Using the PageFilter, read first two rows of the HBase table 'ecommerce_transactions':

```
scan 'ecommerce_transactions', {FILTER => "PageFilter(2)"}
```

Combining this with another filter, get 3 rows with the column family starting with the letter 'c'

```
scan 'ecommerce_transactions', {FILTER => "PageFilter(3) AND ColumnPrefixFilter('c')"}
```

Using the InclusiveStopFilter, scan from start till the specified row 4 (including 4 in it)

```
scan 'ecommerce_transactions', {FILTER => "InclusiveStopFilter('4')"}
```

Now use the ValueFilter to do some basic search on values of the data. This works similar to the 'like' command of SQL. Find the card type starting with 'M'

```
scan 'ecommerce_transactions', {COLUMNS => 'card_type', FILTER =>
"ValueFilter(=,'binary prefix:M')"}
```

Return all countries starting with letters > 'I':

```
scan 'ecommerce_transactions', {COLUMNS => 'countryname', FILTER =>
"ValueFilter(>,'binary prefix:I')"}
```

Using the STARTROW, scan only rows from row key 3 onwards:

```
scan 'ecommerce_transactions', {STARTROW => '3'}
```

A Use Case Combining Hive and HBase

Definition of use case: *To implement HBase for big data storage in NoSQL form and run that with integration to Hive HQL language in order to reuse the existing SQL queries*

Starting HBase shell: hbase shell

Creating table in HBase for e-commerce transactions data:

```
create 'transaction_detail_HBase_tbl','transaction_data','customer_data'
```

Note: the column families are declared in the create table command in order to create it successfully in HBase shell.

Entering some initial values to populate table:

[ToDo: Paste snapshot of below]

```
put 'transaction_detail_HBase_tbl','1','transaction_data:transaction_amount','50.85'

put 'transaction_detail_HBase_tbl','1','transaction_data:transaction_card_type','MasterCard'

put 'transaction_detail_HBase_tbl','1','transaction_data:transaction_ecommerce_website_name','www.ebay.com'

put 'transaction_detail_HBase_tbl','1','transaction_data:transaction_datetime','2019-05-14 15:24:12'

put 'transaction_detail_HBase_tbl','1','transaction_data:transaction_product_name','Laptop'

put 'transaction_detail_HBase_tbl','1','customer_data:transaction_ci ty_name','Mumbai'

put 'transaction_detail_HBase_tbl','1','customer_data:transaction_co untry_name','India'

put 'transaction_detail_HBase_tbl','2','transaction_data:transaction_amount','259.12'

put 'transaction_detail_HBase_tbl','2','transaction_data:transaction_card_type','MasterCard'
```

```
put 'transaction_detail_HBase_tbl','2','transaction_data:transaction_ecommerce_website_name','www.amazon.com'

put 'transaction_detail_HBase_tbl','2','transaction_data:transaction_datetime','2019-05-14 15:24:13'

put 'transaction_detail_HBase_tbl','2','transaction_data:transaction_product_name','Wrist Band'

put 'transaction_detail_HBase_tbl','2','customer_data:transaction_ci ty_name','Pune'

put 'transaction_detail_HBase_tbl','2','customer_data:transaction_co untry_name','India'

put 'transaction_detail_HBase_tbl','3','transaction_data:transaction_amount','328.16'

put 'transaction_detail_HBase_tbl','3','transaction_data:transaction

_card_type','MasterCard'

put 'transaction_detail_HBase_tbl','3','transaction_data:transaction_ecommerce_website_name','www.flipkart.com'

put 'transaction_detail_HBase_tbl','3','transaction_data:transaction_datetime','2019-05-14 15:24:14'

put 'transaction_detail_HBase_tbl','3','transaction_data:transaction_product_name','TV Stand'

put 'transaction_detail_HBase_tbl','3','customer_data:transaction_ci ty_name','New York City'

put 'transaction_detail_HBase_tbl','3','customer_data:transaction_co untry_name','United States'

put 'transaction_detail_HBase_tbl','4','transaction_data:transaction_amount','399.06'

put 'transaction_detail_HBase_tbl','4','transaction_data:transaction_card_type','Visa'

put 'transaction_detail_HBase_tbl','4','transaction_data:transaction_ecommerce_website_name','www.snapdeal.com'

put 'transaction_detail_HBase_tbl','4','transaction_data:transaction_datetime','2019-05-14 15:24:15'
```

put 'transaction_detail_HBase_tbl','4','transaction_data:transaction_product_name','TV Stand'

put 'transaction_detail_HBase_tbl','4','customer_data:transaction_ci ty_name','New Delhi'

put 'transaction_detail_HBase_tbl','4','customer_data:transaction_co untry_name','India'

put 'transaction_detail_HBase_tbl','5','transaction_data:transaction_amount','194.52'

put 'transaction_detail_HBase_tbl','5','transaction_data:transaction_card_type','Visa'

put 'transaction_detail_HBase_tbl','5','transaction_data:transaction_ecommerce_website_name','www.ebay.com'

put 'transaction_detail_HBase_tbl','5','transaction_data:transaction_datetime','2019-05-14 15:24:16'

put 'transaction_detail_HBase_tbl','5','transaction_data:transaction_product_name','External Hard Drive'

put 'transaction_detail_HBase_tbl','5','customer_data:transaction_city_name','Rome'

put 'transaction_detail_HBase_tbl','5','customer_data:transaction_country_name','Italy'

View the populated table to see the rows entered in the previous step:

[ToDo: Paste snapshot of below]

scan 'transaction_detail_HBase_tbl'

Once table has been populated, we exit the HBase shell and enter the Hive shell for integration purposes:

Exiting HBase Shell: exit

Next we enter Hive shell to make a connecting table to our HBase table (Hive docker should also be running)

Enter Hive shell (execute the beeline command)

```
show databases;

create database mydb300;

use mydb300;
```

Now create an external table in hive which points to the table with data kept in HBase following an architecture such that:

[ToDo: Paste snapshot of below]

```
CREATE EXTERNAL TABLE transaction_detail_hive_tbl(transaction_id int,
transaction_card_type string, transaction_ecommerce_website_name string,
transaction_product_name string, transaction_datetime string, transaction_amount
double, transaction_city_name string, transaction_country_name string) STORED BY
'org.apache.hadoop.hive.HBase.HBaseStorageHandler'

WITH SERDEPROPERTIES

("HBase.columns.mapping"=":key,transaction_data:transaction_card_type,transaction_d
ata:transaction_ecommerce_website_name,transaction_data:transaction_product_name
,transaction_data:transaction_datetime,transaction_data:transaction_amount,customer_
data:tran saction_city_name,customer_data:transaction_country_name")

 TBLPROPERTIES

("HBase.table.name"="transaction_detail_HBase_tbl");
```

How this integration works:

A new table is created in Hive database using the create external table command which has all attributes and datatypes declared in Hive in a similar way as SQL tables.

This table is linked with HBase using a stored by operation using the storage handler for HBase as depicted in the architecture diagram above.

Furthermore, this command also adds serdeproperties where we specify all the column families and column family qualifiers that have to be connected to in HBase listing them in the same sequence as listed in the attributes declared after the create external table command.

Lastly this command creates a link to the HBase table using tblproperties.

Note: HBase table may have many columns however, in Hive we do not have to fetch all columns, but just the columns we need for analysis.

Once this is executed, the data can be fetched in Hive using similar queries to SQL.

View the HBase connected table in Hive:

[ToDo: Paste snapshot of below]

select * from transaction_detail_hive_tbl;

Note: all this data is stored in HBase and is directly coming from it. Nothing is ever stored in Hive. Hive is only being used as interface to query the data for simplification purposes.

Inserting Big Data in HBase programmatically:

[ToDo: Do all below with Java/Python]

As seen above, sample data was inserted using the 'put' command in the HBase table to demo the integration with Hive.

However, for big data, multiple put commands will make the data insertion process inefficient. A better approach would be to write a program for data insertion.

Example program code for inserting large data set in HBase (this code is in jRuby, but can be written in java or python as well)

Create a JRuby file named multi.rb through which we will insert multiple rows in an HBase table:

Comment: Setting up packages and function to initiate HBase insertion program

import 'org.apache.hadoop.HBase.client.HTable'

import 'org.apache.hadoop.HBase.client.Put'

```
def jbytes(*args)
        args.map {|arg| arg.to_s.to_java_bytes}
end
```

Comment: Creating HTable as an object called tables pointing to the transaction_detail_HBase_tbl

Comment: Creating an object called "row" with a row key specified

```
table = HTable.new(@HBase.configuration, "transaction_detail_HBase_tbl")
row = Put.new(*jbytes("200"))
```

Comment: Using the add method to put data into the "row" object

```
row.add
(*jbytes("customer_data", "card_type", "master card"))
(*jbytes("customer_data", "card_type", "visa card"))
(*jbytes("customer_data", "card_type", "multi card"))
 table.put(row)
```

Executing from linux command prompt:

```
hbase shell multi.rb
```

Big data import in HBase - Importing data from a CSV file :

Once the table has been created in HBase, then locate the CSV file on your computer. (transaction_detail.csv)

Make sure to remove the header row as attributes are declared in HBase.

[ToDo: Load the CSV file into the Hbase table and paste snapshot]

```
hbase org.apache.hadoop.hbase.mapreduce.ImportTsv - Dimporttsv.separator=',' -
Dimporttsv.columns=HBASE_ROW_KEY,transaction_data:transaction_ca
rd_type,transaction_data:transaction_ecommerce_website_name,tran
saction_data:transaction_product_name,transaction_data:transacti
on_datetime,transaction_data:transaction_amount,customer_data:tr
ansaction_city_name,customer_data:transaction_country_name
transaction_detail_HBase_tbl /user/hdfs/transaction_detail.csv
```

[Run SQL queries on the data using Hive and show snapshot]

Client API: The Basics

Topic: Basics of client API, CRUD operations, KeyValue Class, application for inserting data into database, application for retrieving data from database, and row locks.

To perform CRUD operations which stands for create, read, update, and delete, on HBase table we use Java client API for HBase. Since HBase has a Java Native API and it is written in Java thus it offers programmatic access to DML (Data Manipulation Language).

Class HBase Configuration: This class adds HBase configuration files to a Configuration. It belongs to the org.apache.hadoop.hbase package.

Method: To create a Configuration with HBase resources, we use the following method:

```
static org.apache.hadoop.conf.Configuration create()
```

Class HTable in HBase Client API:

An HBase internal class which represents an HBase table is HTable. Basically, to communicate with a single HBase table, we use this implementation of a table. It belongs to the org.apache.hadoop.hbase.client class.

Constructors: We can create an object to access an HBase table by using the following constructors:

HTable()

HTable(TableName tableName, ClusterConnection connection, ExecutorService pool)

Methods:

void close() : To release all the resources of the HTable, we use this method.

void delete(Delete delete) : The method “void delete(Delete delete)” helps to delete the specified cells/row.

Result get(Get get) : This method retrieves certain cells from a given row.

org.apache.hadoop.conf.Conf iguration getConfiguration() : It returns the Configuration object used by this instance.

TableName getName() : This method returns the table name instance of the table.

HTableDescriptor getTableDescriptor() : It returns the table descriptor for the table.

byte[] getTableName() : This method returns the name of the table.

void put(Put put): We can insert data into the table, by using this method.

Class Put in HBase Client API:

To perform put operations for a single row, we use the class that belongs to the org.apache.hadoop.hbase.client package.

Constructors:

put(byte[] row): We can create a Put operation for the specified row, by using this constructor.

put(byte[] rowArray, int rowOffset, int rowLength): To make a copy of the passed-in row key to keep local, we use it.

put(byte[] rowArray, int rowOffset, int rowLength, long ts): We can make a copy of the passed-in row key to keep local, by using this constructor.

put(byte[] row, long ts): To create a Put operation for the specified row, using a given timestamp, we use it.

Methods

put add(byte[] family, byte[] qualifier, byte[] value): The method “Put add(byte[] family, byte[] qualifier, byte[] value)” adds the specified column and value to this Put operation.

put add(byte[] family, byte[] qualifier, long ts, byte[] value): With the specified timestamp, it adds the specified column and value, as its version to this Put operation.

put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value): This method adds the specified column and value, with the specified timestamp as its version to this Put operation.

put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value): With the specified timestamp, it adds the specified column and value, as its version to this Put operation.

The KeyValue class:

In our code we may have to deal with KeyValue instances directly. These instances contain 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.

The class provides a plethora of constructors that allow you to combine all of these in many variations. The fully specified constructor looks like this: KeyValue(byte[] row, int roffset, int rlength, byte[] family, int foffset, int flength, byte[] qualifier, int qoffset, int qlength, long timestamp, Type type, byte[] value, int voffset, int vlength)

Class Get in HBase Client API:

To perform Get operations on a single row, we use the class that belongs to the org.apache.hadoop.hbase.client package.

Constructor:

get(byte[] row): It is possible to create a Get operation for the specified row, by using this constructor.

Methods:

get addColumn(byte[] family, byte[] qualifier): To retrieve the column from the specific family with the specified qualifier, this method helps.

get addFamily(byte[] family): This one helps to retrieve all columns from the specified family.

Class Delete in HBase Client API

Constructors

delete(byte[] row): To create a delete operation for the specified row, we use it.

delete(byte[] rowArray, int rowOffset, int rowLength): This constructor creates a Delete operation for the specified row and timestamp.

delete(byte[] rowArray, int rowOffset, int rowLength, long ts): This constructor performs Delete operation.

delete(byte[] row, long timestamp): Again this constructor performs Delete operation.

Methods

delete addColumn(byte[] family, byte[] qualifier): This method helps to delete the latest version of the specified column.

delete addColumns(byte[] family, byte[] qualifier, long timestamp): To delete all versions of the specified column we use this method, especially, with a timestamp less than or equal to the specified timestamp.

delete addFamily(byte[] family): This method deletes all versions of all columns of the specified family.

delete addFamily(byte[] family, long timestamp): Again, with a timestamp less than or equal to the specified timestamp, this method also deletes all columns of the specified family.

Class Result in HBase Client API

Constructor

result(): With no KeyValue payload, it is possible to create an empty Result; returns null if you call raw Cells(), by using this constructor.

Methods

byte[] getValue(byte[] family, byte[] qualifier): In order to get the latest version of the specified column, we use this method.

byte[] getRow(): Moreover, to retrieve the row key which corresponds to the row from which this Result was created, we use this method.

Row Locks

The region servers provide a row lock feature ensuring that only a client holding the matching lock can modify a row. In practice, though, most client applications do not provide an explicit lock, but rather rely on the mechanism in place that guards each operation separately.

When you send, for example, a put() call to the server with an instance of Put, created with the following constructor: Put(byte[] row) , Which is not providing a RowLock instance parameter, the servers will create a lock on our behalf, just for the duration of the call.

Example Application:

1. Inserting data into HBase Table Using Java API

```
import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration; import
org.apache.hadoop.hbase.client.HTable; import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes; import java.io.IOException;

public class PutExample {

public static void main(String[] args) throws IOException {


Create the required configuration.

Configuration conf = HBaseConfiguration.create();

Instantiate a new client.

HTable table = new HTable(conf, "testtable");

Create Put with specific row.

Put put = new Put(Bytes.toBytes("row1"));
```

Add a column, whose name is “colfam1:qual1”, to the Put.
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"), Bytes.toBytes("val1"));

Add another column, whose name is “colfam1:qual2”, to the Put.
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual2"), Bytes.toBytes("val2"));

Store the row with the column into the HBase table. table.put(put);

}

}

}

2.Retrieving data from HBase Table Using Java API

Create the configuration:

Configuration conf = HBaseConfiguration.create();

Instantiate a new table reference:

HTable table = new HTable(conf, "testtable");

Create a Get with a specific row:

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

Add a column to the Get:

get.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"));

Retrieve a row with selected columns from HBase:

Result result = table.get(get);

Get a specific value for the given column:

byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"));

Print out the value while converting it back:

System.out.println("Value: " + Bytes.toString(val));

3. Deleting data from HBase Table Using Java API

Create a Delete with a specific row: Delete delete = new Delete(Bytes.toBytes("row1"));

Set a timestamp for row deletes: delete.setTimestamp(1);

Delete a specific version in one column:

delete.deleteColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"), 1);

Delete all versions in one column:

delete.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual1"));

Delete the given and all older versions in one column:

delete.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual3"), 15);

Delete the entire family, all columns and versions:
delete.deleteFamily(Bytes.toBytes("colfam3"));

Delete the given and all older versions in the entire column family, that is, from all columns therein:

delete.deleteFamily(Bytes.toBytes("colfam3"), 3);

Delete the data from the HBase table: table.delete(delete);

table.close();

Scan The HBase Table Using Java API

The getScanner() method of the HTable class can be used to scan the entire data of Hbase table. The getScanner() method expects an instance of the Scan class. To summarize, steps to scan the entire data of HBase table are:

Instantiate the Configuration Class:

Configuration conf = HBaseConfiguration.create();

Instantiate the HTable Class:

HTable table = new HTable(conf, "testtable");

Instantiate the Scan Class: Scan scan = new Scan();

// scan the columns scan.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"));
scan.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("val1"));

Get the ResultScanner instances by calling HTable's getScanner() method:

ResultScanner scanner = table.getScanner(scan); for (Result result = scanner.next(); result != null; result=scanner.next())

System.out.println("Found row : " + result);

Read values from ResultScanner and close it: scanner.close();

Assignment: 2.2% of your grade

[Import your big data into Hbase and query that first through Hbase and then through Hive]

[Show all commands along with important snapshots]