|  |  |
| --- | --- |
| A picture of a winding road and trees  MapReduce Design Pattern Hands on Using Inventory Data:-  [Document subtitle] | Abstract  MapReduce Design Pattern Hands on Using Inventory Data. Different commonly used design pattern running examples  Jose Johny (Big Data Analytics)  Hands on |

# MapReduce Design Pattern Hands on Using Inventory Data:-

MapReduce Design patterns are commonly used techniques to solve problems. This Examples are inspired by most by *MapReduce Design Pattern book by Adam Shook* and *Hadoop Definitive Guide 4th Edition by Tom White.*

What I have done is, taken specific design pattern from the book and created sample program using Inventory Data.

#### Used Cloudera CDH 5.4.2.0

Take a Copy of existing training project in eclipse in CDH copy paste the program from the zip file or import below zip file:-

#### Importing Sample Project:-

In Eclipse🡪File🡪Import🡪General

🡪Existing Project into Workspace

🡪Select Archive Files(browse for the zip file)

🡪Select Projects (InventoryDB)



Browse through InventoryDB🡪src 🡪different packages. Each packages are example one Design pattern

#### Source Files:-

Inventory File:- product\_id,warehouse\_id, quantity\_on\_hand, quantity\_ordered, reorder\_level

Product File:- product\_id, product\_Name

Sales File:- product\_id,Sales\_Order



#### Setting up the Input Directory and putting the files into HDFS:-

a) make dir:-

hadoop fs -mkdir -p /user/cloudera/projects/retail/product

hadoop fs -mkdir -p /user/cloudera/projects/retail/sales

hadoop fs -mkdir -p /user/cloudera/projects/retail/inventory

b) put the files in hdfs:-

hadoop fs -put /home/cloudera/InventoryDB/input/inventory/\* /user/cloudera/projects/retail/inventory

hadoop fs -put /home/cloudera/InventoryDB/input/sales/\* /user/cloudera/projects/retail/sales

hadoop fs -put /home/cloudera/InventoryDB/input/product/\* /user/cloudera/projects/retail/product

#### How to export Jar and Run MapReduce Program:-

InventoryDB🡪RightClick🡪Export🡪Java 🡪Jar File🡪create a path like this in JAR file box (/home/cloudera/InventoryDB/jar/InventoryDb.jar) then 🡪Next🡪Finish (Overwrite if prompting)

#### Running MapReduce:-From the jar folder

hadoop jar <ExportedJarName> <PackageName.DriverClassName> <Args1> <Args2> <Arg3> etc

# Summarization Patterns in MapReduce

The numerical summarizations pattern is a general pattern for calculating aggregate statistical values over your data. Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set.

**Scenario:-Find out the Total Quantity from the Inventory file:-**

Total Inventory = (quantity\_on\_hand + quantity\_ordered) for each product

Mapper 🡪quantity\_on\_hand + quantity\_ordered for each record

Reducer 🡪Sum of above total for each product

**Package:- TotalQuantity**

1. Put the File into HDFS:-

hadoop fs -put /home/cloudera/InventoryDB/input/inventory/\* /user/cloudera/projects/retail/inventory

b) Create Driver, Mapper and Reducer and export the jar:-

/home/cloudera/InventoryDB/jar/ InventoryDb.jar

c) Execute the Hadoop Program:-

hadoop jar InventoryDb.jar TotalQuantity.InvAgg /user/cloudera/projects/retail/inventory /user/cloudera/projects/retail/totalquantity

d) Checking the output:-

hadoop fs -cat /user/cloudera/projects/retail/totalquantity/part-r-00000

# Join Patterns

Data is all over the place, and while it’s very valuable on its own, we can discover interesting relationships when we start analyzing these sets together. This is where **join patterns** come into play. Joins can be used to enrich data with a smaller reference set or they can be used to filter out or select records that are in some type of special list. The use cases go on and on as well.

There different way joining of different datasets achieved.

a) Distributed Cache Join –Can be done in Map Side and Or Reduce Side

Scenario: - Find out the Total Quantity from the Inventory file with product name. Product name for each product can be find by looking up in product data based on product id

**How?-** 🡪Put the Product.txt in Distributed Cache Mapper read the Input Inventory File.

🡪In Mapper setup function read the Distributed Cache file fill the map collection(productMap).

🡪in Mapper , lookup the productMap using product id and get the product Name. Mapper out is Product Name as Key and Total Quantity as Value

🡪Reducer will find the total Quantity for each product

**Create Driver,Mapper ,Reducer and execute below :-**

**Package:- join.dc**

hadoop jar InventoryDb.jar jar join.dc.InvDC /user/cloudera/projects/retail/inventory /user/cloudera/projects/retail/dcjointotalquantity

Output:-

hadoop fs -cat /user/cloudera/projects/retail/dcjointotalquantity/par\*

#### b) Reduce Side Join:-

**Scenario : - - Find out the Total Quantity from the Inventory file with product name.**

**How?** 🡪Use MultipleInputs for reading multiple file and attaching specific Mapper for each.

🡪Product and Inventory Data is streamed through mapper

🡪Attach a key word for value record of each mapper.

Inventory map output is (Product ID, "invrec"+value Part)

Product Map output is (Product ID, "prodrec"+value Part)

🡪Join happens for each key at Reducer Side. Each record in reducer is differentiated based on above flags(invrec or prodrec).

**Create Driver,Mapper ,Reducer and execute below :-**

**Package:-** **join.reducesidejoin**

hadoop jar InventoryDb.jar join.reducesidejoin.InvRSJ /user/cloudera/projects/retail/product /user/cloudera/projects/retail/inventory /user/cloudera/projects/retail/reducejoinout

**output:-**

hadoop fs -cat /user/cloudera/projects/retail/reducejoinout2/part-r-00000

#### Join Assignment:-

Scenario **:-Find the Total Sales of each product.**

Print like (Product Name, Warehouse ID, Sales order).

Product Name from product file, Warehouse Id from Inventory file, Sales Order from Sales file.

One method:- Put Distributed Cache for below files

Cache file ->/user/cloudera/projects/retail/product/Product.txt#Product.txt

Cache file ->/user/cloudera/projects/retail/sales/Sales.txt#Sales.txt

Stream the Inventory File:- Use Inventory Mapper for it (Mapper Inventory)

/user/cloudera/projects/retail/inventory/

2) Create Driver Mapper and Reducer

**Package:- TotalSalesQoH**

3) Export the Jar and execute the MapReduce program

a) Exported Jar location:-

/home/cloudera/InventoryDB/jar/TotalSalesQoH.jar

b) Execute Command:-

hadoop jar InventoryDb.jar TotalSalesQoH.DriverTotalSalesQoH /user/cloudera/projects/retail/inventory/ /user/cloudera/projects/retail/totalsales

# Use of Partitioning in MapReduce

The partitioning pattern moves the records into categories (i.e., shards, partitions, or bins) but it doesn’t really care about the order of records. The intent is to take similar records in a data set and partition them into distinct, smaller data sets.

**Scenario : -** We have Inventory file with Warehouse details. Partition the data based on Warehouse ID.

InputFile:-



How? 🡪Create a Custom Partitioner class with WareHouse ID as the Key

🡪Use this in Driver class and also set no of reduce tasks

🡪 Inventory files are divided into the No of reduce tasks set in the Driver

**Package:- partition**

hadoop jar InventoryDb.jar partition.InvPartitionDriver /user/cloudera/projects/retail/inventory2/ /user/cloudera/projects/retail/WareHousePartitionedData

**OutPut:-**

hadoop fs -ls /user/cloudera/projects/retail/WareHousePartitionedData/

-rw-r--r-- 1 cloudera cloudera 0 2016-04-13 03:53 /user/cloudera/projects/retail/WareHousePartitionedData/part-r-00000

-rw-r--r-- 1 cloudera cloudera 0 2016-04-13 03:53 /user/cloudera/projects/retail/WareHousePartitionedData/part-r-00001

-rw-r--r-- 1 cloudera cloudera 0 2016-04-13 03:54 /user/cloudera/projects/retail/WareHousePartitionedData/part-r-00002

-rw-r--r-- 1 cloudera cloudera 817 2016-04-13 03:54 /user/cloudera/projects/retail/WareHousePartitionedData/part-r-00003

# Binning Pattern:-

The binning pattern, much like the partitioning pattern, moves the records into categories irrespective of the order of records. We want to filter data by tag into different bins so that we can run follow-on analysis without having to run over all of the data.

Using MapReduce **MultipleOutputs** options you can do binning at Mapper output or Reducer output.

Scenario:-

Divide the Inventory records based on Warehouse details. Output will be one file for each warehouse.

**Package:- CustomInpFormat**

hadoop jar InventoryDb.jar CustomInpFormat.DriverInventory /user/cloudera/projects/retail/inventory2 /user/cloudera/projects/retail/CustomOutPut

Output Example:-

hadoop fs -ls /user/cloudera/projects/retail/CustomOutPut

Found 6 items

-rw-r--r-- 1 cloudera cloudera 864 2016-04-14 21:24 /user/cloudera/projects/retail/CustomOutPut/WH01-r-00000

-rw-r--r-- 1 cloudera cloudera 540 2016-04-14 21:24 /user/cloudera/projects/retail/CustomOutPut/WH02-r-00000

-rw-r--r-- 1 cloudera cloudera 540 2016-04-14 21:24 /user/cloudera/projects/retail/CustomOutPut/WH03-r-00000

-rw-r--r-- 1 cloudera cloudera 378 2016-04-14 21:24 /user/cloudera/projects/retail/CustomOutPut/WH04-r-00000

# Total order sorting Pattern:-

The total order sorting pattern is concerned with the order of the data from record to record. You want to sort your data in parallel on a sort key. Sorting is easy in sequential programming. Sorting in MapReduce, or more generally in parallel, is not easy. This is because the typical “divide and conquer” approach is a bit harder to apply here.

This pattern has two phases: an analyze phase that determines the ranges, and the order phase that actually sorts the data. The analyze phase is a random sampling of the data. The partitions are then based on that random sample.

First have to determine a set of partitions divided by ranges of values that will produce equal-sized subsets of data. These ranges will determine which reducer will sort which range of data. Then something similar to the partitioning pattern is run: a custom partitioner is used to partition data by the sort key. The lowest range of data goes to the first reducer, the next range goes to the second reducer, so on and so forth.

**Steps:-**

1. 1st created a Sequencefile using job1. Hadoop’s sequence file format stores sequences of binary key-value pairs.
2. Job2 uses InputSampler to sample the file and create partition file
3. Job2 uses the partition file in TotalOrderPartitioner and order the data based on partitioned file
4. Reducer create the ordered output file

InputSampler.writePartitionFile(orderJob, new InputSampler.RandomSampler(sampleRate, 2)

We have total around 40 records, so 2 reducer, No of samples to 2 and sampling frequency 10 records

**Package:- TotalOrderSorting**

hadoop jar InventoryDb.jar TotalOrderSorting.InvTotalOrderSorting /user/cloudera/projects/retail/inventory2 /user/cloudera/projects/retail/totalorder 10

hadoop fs -ls /user/cloudera/projects/retail/totalorder

Found 3 items

-rw-r--r-- 1 cloudera cloudera 0 2016-04-14 23:36 /user/cloudera/projects/retail/totalorder/\_SUCCESS

-rw-r--r-- 1 cloudera cloudera 361 2016-04-14 23:36 /user/cloudera/projects/retail/totalorder/part-r-00000

-rw-r--r-- 1 cloudera cloudera 456 2016-04-14 23:36 /user/cloudera/projects/retail/totalorder/part-r-00001

# Secondary Sorting:-

If you want to have a primary sort key and a secondary sort key, concatenate the keys, delimited by something. For example, if you want to sort by last name first, and city second, use a key that looks like Smith^Baltimore.

Scenario: - Sort by product id and warehouse id

//Secondary Sorting: - by product\_id then by wareHouseId

context.write(new Text(product\_id+"^"+wareHouseId), value);

**Package:- SecondarySorting**

hadoop jar InventoryDb.jar SecondarySorting.InvSecondarySorting /user/cloudera/projects/retail/inventory2 /user/cloudera/projects/retail/SecondarySorting 10

Output Sample:-

hadoop fs -ls /user/cloudera/projects/retail/SecondarySorting

Found 3 items

-rw-r--r-- 1 cloudera cloudera 0 2016-04-15 00:15 /user/cloudera/projects/retail/SecondarySorting/\_SUCCESS

-rw-r--r-- 1 cloudera cloudera 684 2016-04-15 00:15 /user/cloudera/projects/retail/SecondarySorting/part-r-00000

-rw-r--r-- 1 cloudera cloudera 133 2016-04-15 00:15 /user/cloudera/projects/retail/SecondarySorting/part-r-00001

hadoop fs -cat /user/cloudera/projects/retail/SecondarySorting/part\*00000

1001,WH01,68,25,30

1001,WH02,68,25,30

1001,WH03,68,25,30

1002,WH01,35,50,15

1002,WH02,35,50,15

1002,WH03,35,50,15

Another way is create a Custom Key class with natural key and natural value. Then setSortComparatorClass to do set the comparator. In the comparator should order the Custom Key (natural key and natural value).

Check Hadoop definitive Guide 4th edition 🡪Secondary Sort

# Customizing Input format in Hadoop

**Scenario: -**

All the above examples, we were using TextInputFormat to read input files. If we want to read parse the any file, we can create custom input format. So any file (Json, Weblog etc.) can be read using MapReduce.

Hadoop relies on the input format of the job to do three things:

1. Validate the input configuration for the job (i.e., checking that the data is there).

2. Split the input blocks and files into logical chunks of type InputSplit, each of which is assigned to a map task for processing.

3. Create the RecordReader implementation to be used to create key/value pairs from the raw InputSplit. These pairs are sent one by one to their mapper.

How? 🡪Created Custom Key and Value Class

🡪Created a custom recordReader by extending RecordReader

🡪 Created custom Input fomat using above record reader and extending FileInputFormat . It emit Custom Key and Value Object. Which goes to the Mapper

**Package: - CustomInpFormat**

hadoop jar InventoryDb.jar CustomInpFormat.DriverInventory /user/cloudera/projects/retail/inventory /user/cloudera/projects/retail/CustomOutPut

# Filtering Pattern:-

Filtering simply evaluates each record separately and decides, based on some condition, whether it should stay or go.

Simple Random Sampling: - In simple random sampling (SRS), we want to grab a subset of our larger data set in which each record has an equal probability of being selected. Typically this is useful for sizing down a data set to be able to do representative analysis on a more manageable set of data.

Package:- SimpleRandomSampling

hadoop jar InventoryDb.jar SimpleRandomSampling.InvSimpleRandomSampling /user/cloudera/projects/retail/inventory2 /user/cloudera/projects/retail/SRSoutput10 11

#### Bloom Filtering Pattern: -

A Bloom filter is a probabilistic data structure used to test whether a member is an element of a set. Bloom filters have a strong space advantage over other data structures such as a Java Set, in that each element uses the same amount of space, no matter its actual size.

While the data structure itself has vast memory advantages, it is not always 100% accurate.

While **false positives are possible**, **false negatives are not**. This means the result of each test is either a definitive “no” or “maybe.” You will never get a definitive “yes.”

**Training the Bloom Filter:-**

m🡪The number of bits in the filter

n🡪The number of members in the set

p🡪The desired false positive rate

k🡪The number of different hash functions used to map some element to one of the mbits with a uniform random distribution.

Optimal size of a Bloom

filter with an optimal-k = 

Optimal-k of a Bloom filter = 

Create the Bloom filter file and save in hdfs: - we selected particular products and training the bloom filter with it.

**Package:- BloomFiltering**

hadoop jar InventoryDb.jar BloomFiltering.BloomFilterDriver /user/cloudera/projects/retail/bloomfilter/Product\_BloomInput.txt 5 0.1 /user/cloudera/projects/retail/bloomfilter/Product\_BloomInput.bf

Use the above created bloom filter to filter out the products from inventory file.

hadoop jar InventoryDb.jar BloomFiltering.Product\_BloomFilteringDriver /user/cloudera/projects/retail/inventory2/ /user/cloudera/projects/retail/bloomFilteroutput

# Data Organization Patterns:-

**Structured to Hierarchical:-**

The structured to hierarchical pattern creates new records from data that started in a very different structure. This pattern used for transforming your row-based data to a hierarchical format, such as JSON or XML.

When migrating data from an RDBMS to a Hadoop system, one of the first things you should consider doing is reformatting your data into a more conducive structure. Since Hadoop doesn’t care what format your data is in, you should take advantage of hierarchical data to avoid doing joins.

#### Structured to Hierarchical Examples:-Create XML file

Scenario:-Combine the Inventory and Product File, then create a structured XML hierarchy to nest product name with inventory details

Product1

Inventory1

Inventory2

Product2

Inventory1

Inventory2

**XML creation:-**We used DOM XML parser to create a XML file:-

DOM provides many handy classes to create XML file easily. Firstly, you have to create a Document with DocumentBuilder class, define all the XML content – node, attribute with Element class. In last, use Transformer class to output the entire XML content to stream output, typically a File.

**Package:- buildingHierarchical.xmlfiles**

hadoop jar InventoryDb.jar buildingHierarchical.xmlfiles.InvRSJ /user/cloudera/projects/retail/product /user/cloudera/projects/retail/inventory /user/cloudera/projects/retail/xmlfiles5

hadoop fs -cat /user/cloudera/projects/retail/xmlfiles5/part-r-00000

#### Structured to Hierarchical Examples:-Create Json File

JSON is a lightweight data-interchange format that is widely used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet. There are different API for creating Json file.

I have used Gson jar for creating Json file from Custom object. It convert the object into Json record.

Downloaded this **gson-2.6.2.jar.** Then put the Gson jar into the Distributed Cache for Mapper to use this.

try{DistributedCache.addFileToClassPath(new Path("/user/cloudera/projects/WebLogAnalySiS/externalJars/gson-2.6.2.jar"), job.getConfiguration()); }catch(Exception e){

System.out.println(e);

}

job.setOutputFormatClass(TextOutputFormat.class);

#### Mapper Code Sample for creating Json file

import java.io.IOException;

import org.apache.hadoop.io.NullWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

public class WebLogMapper extends Mapper<Text, WebLogWritable, NullWritable, Text> {

protected void map(Text key, WebLogWritable value, Context context) throws IOException, InterruptedException {

// String[] tokens = value.toString().split(",");

**Gson gson = new GsonBuilder().disableHtmlEscaping().create();**

**// convert java object to JSON format,**

**// and returned as JSON formatted string**

**String json = gson.toJson(value)** ;

//context.write(NullWritable.get(), new Text(json));

context.write(NullWritable.get(), new Text(json));

}

}

# References:-

**🡪MapReduce Design Pattern book by Adam Shook**

**🡪Hadoop Definitive Guide 4th Edition by Tom White**

**Thank you**

**Jose Johny**