**MapReduce Example: Reduce Side Join in Hadoop MapReduce**

**Introduction:**

In this blog, I am going to explain you how a reduce side join is performed in Hadoop MapReduce using a MapReduce example. Here, I am assuming that you are already familiar with MapReduce framework and know how to write a basic MapReduce program. In case you don’t, I would suggest you to go through my previous blog on [***MapReduce Tutorial***](http://www.edureka.co/blog/mapreduce-tutorial/?utm_source=blog&utm_medium=content-link&utm_campaign=mapreduce-example-reduce-side-join) so that you can grasp the concepts discussed here without facing any difficulties. The topics discussed in this blog are as follows:

* What is a Join?
* Joins in MapReduce
* What is a Reduce side join?
* MapReduce Example on Reduce side join
* Conclusion

**What is a Join?**

The join operation is used to combine two or more database tables based on foreign keys. In general, companies maintain separate tables for the customer and the transaction records in their database. And, many times these companies need to generate analytic reports using the data present in such separate tables. Therefore, they perform a join operation on these separate tables using a common column (foreign key), like customer id, etc., to generate a combined table. Then, they analyze this combined table to get the desired analytic reports.

**Joins in MapReduce**

Just like SQL join, we can also perform join operations in MapReduce on different data sets. There are two types of join operations in MapReduce:

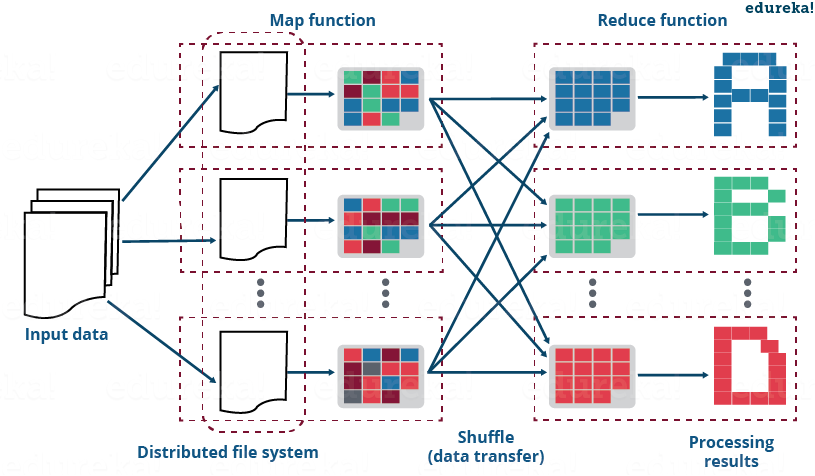
* **Map Side Join:** As the name implies, the join operation is performed in the map phase itself. Therefore, in the map side join, the mapper performs the join and it is mandatory that the input to each map is partitioned and sorted according to the keys.

The map side join has been covered in a separate blog with an example. [***Click Here***](http://www.edureka.co/blog/map-side-join-vs-join/?utm_source=blog&utm_medium=content-link&utm_campaign=mapreduce-example-reduce-side-join) to go through that blog to understand how the map side join works and what are its advantages.

* **Reduce Side Join:** As the name suggests, in the reduce side join, the reducer is responsible for performing the join operation. It is comparatively simple and easier to implement than the map side join as the sorting and shuffling phase sends the values having identical keys to the same reducer and therefore, by default, the data is organized for us.

Now, let us understand the reduce side join in detail.

**What is Reduce Side Join?**



As discussed earlier, the reduce side join is a process where the join operation is performed in the reducer phase. Basically, the reduce side join takes place in the following manner:

* Mapper reads the input data which are to be combined based on common column or join key.
* The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases.
* The mapper outputs the intermediate key-value pair where the key is nothing but the join key.
* After the sorting and shuffling phase, a key and the list of values is generated for the reducer.
* Now, the reducer joins the values present in the list with the key to give the final aggregated output.

*Meanwhile, you may go through this MapReduce Tutorial video where various MapReduce Use Cases has been clearly explained and practically demonstrated:*

**MapReduce Example | MapReduce Programming | Hadoop MapReduce Tutorial | Edureka**

[Learn MapReduce From Industry Experts](http://www.edureka.co/big-data-and-hadoop?utm_source=blog&utm_medium=blog-cta&utm_campaign=mapreduce-example-reduce-side-join)

Now, let us take a MapReduce example to understand the above steps in the reduce side join.

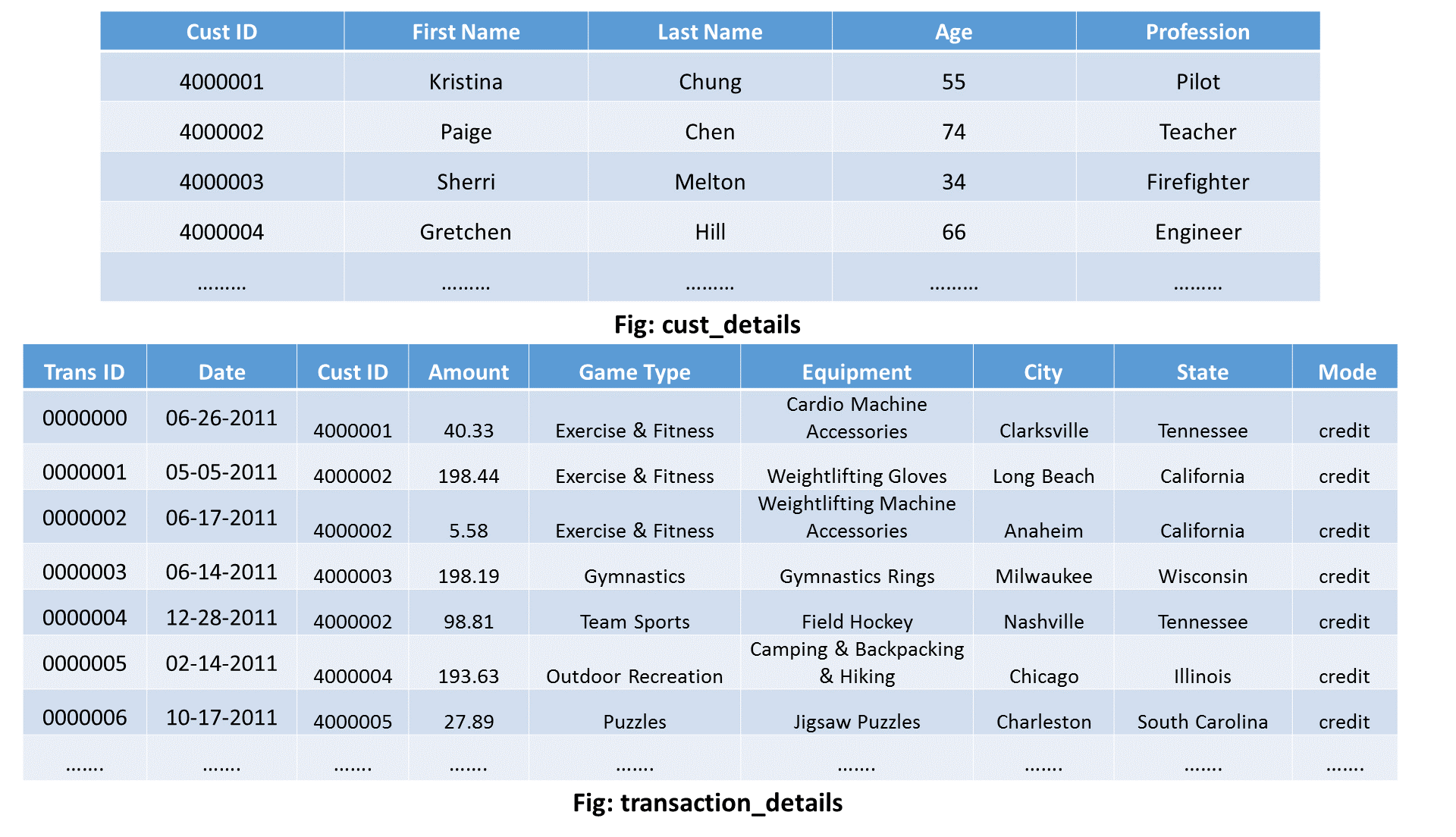
**https://ssl.gstatic.com/ui/v1/icons/mail/images/cleardot.gifMapReduce Example of Reduce Side Join**

Suppose that I have two separate datasets of a sports complex:

* **cust\_details:** It contains the details of the customer.
* **transaction\_details:** It contains the transaction record of the customer.

Using these two datasets, I want to know the lifetime value of each customer. In doing so, I will be needing the following things:

* The person’s name along with the frequency of the visits by that person.
* The total amount spent by him/her for purchasing the equipment.



The above figure is just to show you the schema of the two datasets on which we will perform the reduce side join operation. Click on the button below to download the whole project containing the source code and the input files for this MapReduce example:

[Download](https://goo.gl/IeZZmo?utm_source=blog&utm_medium=button&utm_campaign=mapreduce-example-reduce-side-join-download)

Kindly, keep the following things in mind while importing the above MapReduce example project on reduce side join into Eclipse:

* The input files are in input\_files directory of the project. Load these into your HDFS.
* Don’t forget to build the path of Hadoop Reference Jars (present in reduce side join project lib directory)  according to your system or VM.

Now, let us understand what happens inside the map and reduce phases in this MapReduce example on reduce side join:

**1. Map Phase:**

I will have a separate mapper for each of the two datasets i.e. One mapper for cust\_details input and other for transaction\_details input.

***Mapper for cust\_details:***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static class CustsMapper extends Mapper <Object, Text, Text, Text>  {  public void map(Object key, Text value, Context context) throws IOException, InterruptedException  {String record = value.toString();  String[] parts = record.split(",");  context.write(new Text(parts[0]), new Text("cust\t" + parts[1]));  }  } |

* I will read the input taking one tuple at a time.
* Then, I will tokenize each word in that tuple and fetch the cust ID along with the name of the person.
* The cust ID will be my key of the key-value pair that my mapper will generate eventually.
* I will also add a tag “cust” to indicate that this input tuple is of cust\_details type.
* Therefore, my mapper for cust\_details will produce following intermediate key-value pair:

**Key – Value pair: [cust ID, cust        name]**

Example: [4000001, cust    Kristina], [4000002, cust   Paige], etc.

***Mapper for transaction\_details:***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public static class TxnsMapper extends Mapper <Object, Text, Text, Text>  {  public void map(Object key, Text value, Context context) throws IOException, InterruptedException  {  String record = value.toString();  String[] parts = record.split(",");  context.write(new Text(parts[2]), new Text("tnxn\t" + parts[3]));  }  } |

* Like mapper for cust\_details, I will follow the similar steps here. Though, there will be a few differences:
  + I will fetch the amount value instead of name of the person.
  + In this case, we will use “tnxn” as a tag.
* Therefore, the cust ID will be my key of the key-value pair that the mapper will generate eventually.
* Finally, the output of my mapper for transaction\_details will be of the following format:

**Key, Value Pair: [cust ID, tnxn   amount]**

*Example:* [4000001, tnxn   40.33], [4000002, tnxn   198.44], etc.

**2. Sorting and Shuffling Phase**

The sorting and shuffling phase will generate an array list of values corresponding to each key. In other words, it will put together all the values corresponding to each unique key in the intermediate key-value pair. The output of sorting and shuffling phase will be of the following format:

**Key – list of Values:**

* {cust ID1 – [(cust    name1), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* {cust ID2 – [(cust    name2), (tnxn    amount1), (tnxn    amount2), (tnxn    amount3),…..]}
* ……

**Example:**

* {4000001 – [(cust    kristina), (tnxn    40.33), (tnxn    47.05),…]};
* {4000002 – [(cust    paige), (tnxn    198.44), (tnxn     5.58),…]};
* ……

Now, the framework will call reduce() method (*reduce(Text key, Iterable<Text> values, Context context)*) for each unique join key (cust id) and the corresponding list of values. Then, the reducer will perform the join operation on the values present in the respective list of values to calculate the desired output eventually. Therefore, the number of reducer task performed will be equal to the number of unique cust ID.

Let us now understand how the reducer performs the join operation in this MapReduce example.

**3. Reducer Phase**

If you remember, the primary goal to perform this reduce-side join operation was to find out that how many times a particular customer has visited sports complex and the total amount spent by that very customer on different sports. Therefore, my final output should be of the following format:

**Key – Value pair: [Name of the customer] (Key) – [total amount, frequency of the visit] (Value)**

***Reducer Code:***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>  {  public void reduce(Text key, Iterable<Text> values, Context context)  throws IOException, InterruptedException  {  String name = "";  double total = 0.0;  int count = 0;  for (Text t : values)  {  String parts[] = t.toString().split("\t");  if (parts[0].equals("tnxn"))  {  count++;  total += Float.parseFloat(parts[1]);  }  else if (parts[0].equals("cust"))  {  name = parts[1];  }  }  String str = String.format("%d\t%f", count, total);  context.write(new Text(name), new Text(str));  }  } |

So, following steps will be taken in each of the reducers to achieve the desired output:

* In each of the reducer I will have a key & list of values where the key is nothing but the cust ID. The list of values will have the input from both the datasets i.e. Amount from transaction\_details and name from cust\_details.
* Now, I will loop through the values present in the list of values in the reducer.
* Then, I will split the list of values and check whether the value is of transaction\_details type or cust\_details type.
* If it is of the transaction\_details type, I will perform the following steps:
  + I will increase the counter value by one to calculate the frequency of visit by the very person.
  + I will cumulatively update the amount value to calculate the total amount spent by that person.
* On the other hand, if the value is of cust\_details type, I will store it in a string variable. Later, I will assign the name as my key  in my output key-value pair.
* Finally, I will write the output key-value pair in the output folder in my HDFS.

Hence, the final output that my reducer will generate is given below:

**Kristina, 651.05 8**

**Paige, 706.97  6**

**…..**

And, this whole process that we did above is called ***Reduce Side Join*** in MapReduce.

**Source Code:**

The source code for the above MapReduce example of the reduce side join is given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.Reducer;  import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;     public class ReduceJoin {   public static class CustsMapper extends Mapper <Object, Text, Text, Text>   {   public void map(Object key, Text value, Context context)   throws IOException, InterruptedException   {   String record = value.toString();   String[] parts = record.split(",");   context.write(new Text(parts[0]), new Text("cust\t" + parts[1]));   }   }     public static class TxnsMapper extends Mapper <Object, Text, Text, Text>   {   public void map(Object key, Text value, Context context)   throws IOException, InterruptedException   {   String record = value.toString();   String[] parts = record.split(",");   context.write(new Text(parts[2]), new Text("tnxn\t" + parts[3]));   }   }     public static class ReduceJoinReducer extends Reducer <Text, Text, Text, Text>   {   public void reduce(Text key, Iterable<Text> values, Context context)   throws IOException, InterruptedException   {   String name = "";   double total = 0.0;   int count = 0;   for (Text t : values)   {   String parts[] = t.toString().split("\t");   if (parts[0].equals("tnxn"))   {   count++;   total += Float.parseFloat(parts[1]);   }   else if (parts[0].equals("cust"))   {   name = parts[1];   }   }   String str = String.format("%d\t%f", count, total);   context.write(new Text(name), new Text(str));   }   }     public static void main(String[] args) throws Exception {   Configuration conf = new Configuration();   Job job = new Job(conf, "Reduce-side join");   job.setJarByClass(ReduceJoin.class);   job.setReducerClass(ReduceJoinReducer.class);   job.setOutputKeyClass(Text.class);   job.setOutputValueClass(Text.class);     MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, CustsMapper.class);   MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, TxnsMapper.class);   Path outputPath = new Path(args[2]);     FileOutputFormat.setOutputPath(job, outputPath);   outputPath.getFileSystem(conf).delete(outputPath);   System.exit(job.waitForCompletion(true) ? 0 : 1);   }   } |

**Run this Program**

Finally, the command to run the above MapReduce example program on reduce side join is given below:

*hadoop jar reducejoin.jar ReduceJoin /sample/input/cust\_details /sample/input/transaction\_details /sample/output*

**Conclusion:**

The reduce side join procedure generates a huge network I/O traffic in the sorting and reducer phase where the values of the same key are brought together. So, if you have a large number of different data sets having millions of values, there is a high chance that you will encounter an OutOfMemory Exception i.e. Your RAM is full and therefore, overflown. In my opinion, the advantages of using reduce side join are:

* It is very easy to implement as we are taking advantage of the inbuilt sorting and shuffling algorithm in the MapReduce framework which combines values of the same key and send it to the same reducer.
* In the reduce side join, your input does not require to follow any strict format and therefore, you can perform the join operation on unstructured data as well.

In general, people prefer Apache Hive, which is a part of the Hadoop ecosystem, to perform the join operation. So, if you are from the SQL background, you don’t need to worry about writing the MapReduce Java code for performing a join operation. You can use Hive as an alternative.

[<< Previous Blog](http://www.edureka.co/blog/mapreduce-tutorial?utm_source=blog&utm_medium=blog-cta&utm_campaign=mapreduce-example-reduce-side-join-p)   [Next](https://www.edureka.co/blog/introduction-to-apache-hive/?utm_source=blog&utm_medium=blog-cta&utm_campaign=mapreduce-example-reduce-side-join-n)