*Research Journal of Computer and Information Technology Sciences* _____E-ISSN 2320 – 6527
Vol. **4(9),** 1-4, September **(2016)** *Res. J. Computer and IT Sci.*

***Case Study***

# Analysing Cascading over MapReduce

**Kaustuv Kunal**
Big Data Analyst and Independent Researcher, Delhi, India
kaustuv.kunal@gmail.com

**Available online at: www.isca.in, www.isca.me**
Received 26th June 2016, revised 15th September 2016, accepted 17th September 2016

## Abstract

*In recent years Big Data has grown significantly. Hadoop has become a de-facto Big Data technology and Map-Reduce de-facto processing framework. Hadoop with MapReduce performs distributed processing of large data sets in fault tolerant and cost effective manner. Cascading is an abstraction layer upon MapReduce and allows developers to think with reference to tuples and fields. Many business problems can be solved conveniently with tuple rather than MapReduce key-value pair. The paper advocates cascading over MapReduce and illustrates how lengthy tasks in MapReduce are easily done in Cascading supported by a case study.*

**Keywords:** Big-Data, Cascading, MapReduce, Hadoop, Avro.

## Introduction

The digital data is increasing exponentially. 90% of online data arrived in the past two years. Traditional systems of data processing are not sufficient enough to process scaling data sets. Google published paper on Google File System in 2003[1]. It advocated use of large block sizes for processing large datasets. The paper was followed by MapReduce paper in 2004[2]. MapReduce is data processing frameworks based on divide and conquer approach. These papers were quite intriguing to big data community.

Based on GFS and MapReduce Hadoop was created[3]. Later Hadoop became one of apache open source project[4]. Analytics & Internet scale business companies found Hadoop very useful. Lots of them started using Hadoop for data processing and eventually started contributing to the project. Few organizations started offering their proprietary Hadoop distribution. Cloudera, Hortonworks, IBM BigInsights, MapR are prominent Hadoop distributions[5-8].

MapReduce divides the problem into two phases, map and reduce[9]. Input and output of each phase is key-value pair. Decomposing a problem into key and value is not always feasible. MapReduce framework is complex. Features like record reader, inputs & outputs format, writable data type increases programming overhead and adversely effects code reusability and understanding. Furthermore, MapReduce require strict data preparation, which should be performed outside MapReduce. Most real time business problems require multiple MapReduce chained together. Cascading was developed to address these issues[10].

Cascading provides the means to prepare and manage your data as integral part of program. It architects the problem in terms of source, tap, flow and sink. The data flows in taps as tuple consist of fields.

The paper presents a comparative analysis of cascading and explains its advantages over plain MapReduce using a case study. Cascading introduction & features are discussed in section 2. Section 3 presents comparative analysis between cascading and MapReduce followed up with case study in section 4. The paper concludes in Section 5.

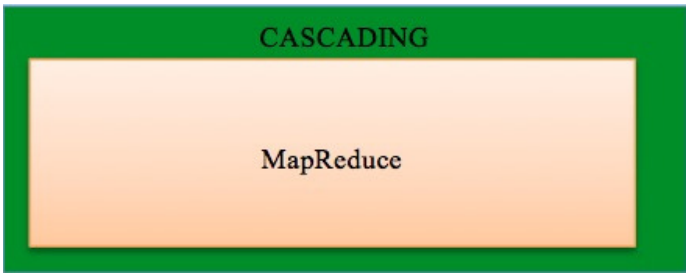


**Figure-1**
**Cascading: An abstraction over MapReduce**

## Cascading Overview

Authored by Chris Wensel, Cascading is a java-processing library. Like MapReduce, Cascading is also a divide and conquer approach but with a different metaphor. It thinks of user data as stream and uses taps, pipes and sinks abstraction. Data is stream of tuple. Tuple is ordered list of fields. Tuples travel inside pipes from sink. Pipes allow parallel execution. Combination of pipes crates a flow. Cascading job is DAG of flow where MapReduce operations are abstracted behind one or more pipe instance. Cascading abstract away keys and values and replace them with tuples that have corresponding field

names similar in concept to table and column names in relational databases.

Cascading concepts and terms can be summarized as follows. Scheme; converts raw data to a tuple and vice versa. Tap; All data is written to tap instance. SinkTap; Specifies Output. Three sinkmode are available, for keeping, replacing/updating and appending. Pipe; perform data analysis. Pipe assemblies are basically Directed Acyclic Graphs (DAG's). Flow; is data processing pipeline to firstly read data from sources, secondly processes the data and finally writes processed output to the sinks. Cascading has support for Custom Operations, Built-in Functions, Built in Assemblies and Fields Algebra. The details are available in cascading user guide.

## Cascading over MapReduce

Cascading is an abstraction over MapReduce. Coupling happens at the very end in cascading, this allows late binding hence minimal dependency. Few of its advantages over MapReduce are.

## Secondary Sort

To implement secondary sort in MapReduce developer needs to implement map, reduce, composite key, a value, a partitioner, output value, grouping comparator, output key comparator and then couple all with each other. Moreover this code would not be reusable one, though in cascading secondary sort is performed with just one line of code.

new GroupBy( <previous-pipe>,<grouping-fields>,<secondry-sort-fields>, <pipe>) ;

## Small Files and combine input format

To process small files in Hadoop, developer has to implement combine input format. This further requires customizing InputFormat, Record reader, writable along with mapper, reducer and driver with no guarantee of them being reused. Inside Cascading it is just a matter of setting properties to enable combine file input format. Method for setting Cascading job properties is shown in Appendix-1.
properties.put("HfsProps.setUseCombinedInput", true);

**Input support:** Cascading provides various utilities to process lot of input type including Avro and JSON.

**Debug and Test:** On single node cluster, Cascading's local mode can be used to efficiently test code and process local files before being deployed on a distributed cluster. Specific debug functions allow convenient way of logging and debugging messages.

**Case Study – Advertiser Insights:** Advertisers need their campaign insights to tune the campaign properly. Data generated from campaigns are large enough and suitable to be processed using Big Data technologies like MapReduce. On the most basic level advertisers need two kind of information first, total number of users or hits and second, total number of unique users or unique visitors.

We converted campaign logs into Avro to provide schema[11]. Avro conversion allows serialization hence faster execution also it provided independence from future modifications. We processed Avro logs via MapReduce. In MapReduce we had to customize a lot of classes starting from key, value pair, record reader, input and output format. This resulted into high code complexity and low reusability quotient.

Later we hear about cascading and decided to give it a try. In cascading we required two kinds of jobs count and count unique. Count is normal aggregation job e.g. finding number of hits. Count unique is cumulative jobs e.g. number of unique user. Results were positive. With Cascading, we developed count and count unique jobs much faster and with shorter testing cycle. A sample of count and count jobs are given in Appendix 2 and Appendix 3 respectively. We found cascading more suitable for processing these kinds of jobs than MapReduce because of its SQL friendly approach. Figure-2 depicts reduction in line of codes (LOC), from MapReduce to Cascading for different jobs.

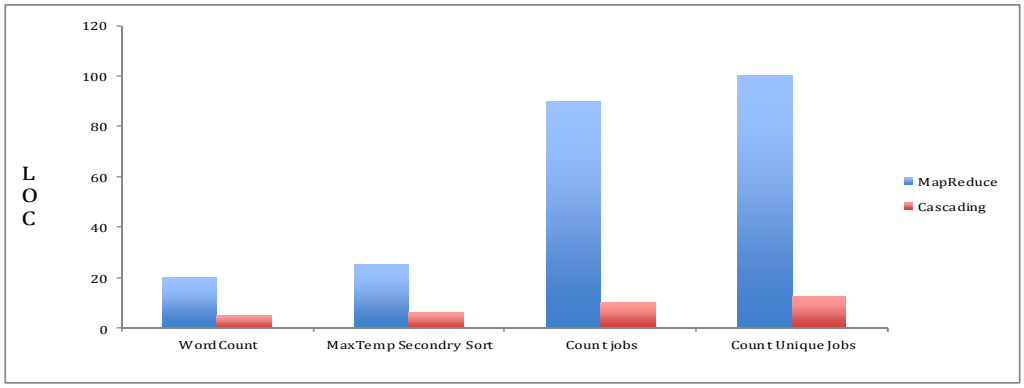


**Figure-2**
**LOC (lines of code) Comparison MapReduce vs Cascading**

## Conclusion

We have advocated Cascading over plain MapReduce programing. Cascading simplifies Hadoop application development process including job creation and job management and scheduling by adding an abstraction layer over Hadoop API. In code testing, code maintenance, enhancements and peer review cascading performs better than MapReduce and allows developers to focus on business logic. Cascading provide freedom from chaining subsequent map reduce. Cascading builds application faster. We shifted our code base from MapReduce to cascading and found that Creating and maintaining MapReduce jobs via cascading is lot easier.

## References

1. Ghemawat Sanjay, Gobioff Howard and Leung Shun-Tak (2003). The Google File System. Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA. October 19-22.
2. Dean Jeffrey and Ghemawat Sanjay (2004). MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, December.
3. Tom White (2012. Hadoop: The definitive guide. O'Reilly Media publication, ISBN-13: 978-1491901632.
4. Hadoop (2016). What Is Apache Hadoop?. Hadoop, http://hadoop.apache.org/, June 21, 2016.
5. Cloudera (2016). Apache Hadoop. Cloudera, https://cloudera.com/products/apache-hadoop.html, June 21, 2016
6. Hortonworks (2016). Hadoop. http://hortonworks.com, June 21, 2016.
7. IBM (2016). Hadoop: Built for big data, insights, and innovation. IBM, USA, http://www.ibm.com/analytics/us/en/technology/hadoop/, access June 21, 2016
8. MapR (2016). Hadoop. MAPR, https://www.mapr.com/, access June 21, 2016.
9. Hadoop (2016). Apache MapReduce. Hadoop, https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html, June 21, 2016
10. Cascading (2016). MapReduce. Cascading, http://www.cascading.org/, June 21, 2016
11. Apache Avro (2016). Avro. Apache Avro, https://avro.apache.org/, June 21, 2016.

**Appendix 1: Setting up cascading properties**

```
Map<Object, Object> getPropertiesForCascadingJob()
  {
      Map<Object, Object> properties = new HashMap<>();
      properties.put("mapred.reduce.tasks.maximum", 15);
      properties.put("mapred.tasktracker.map.tasks.maximum",
40);

      properties.put("fs.s3n.awsAccessKeyId",
AWS_ACCESS_KEY_ID);
      properties.put("fs.s3n.awsSecretAccessKey",
AWS_SECRET_KEY);
      properties.put("HfsProps.setUseCombinedInput", true);
      return properties;
  }
```

**Appendix 2: Cascading Count Jobs**

```
{ //Count jobs

      // Source tap
      Set<Tap> sourceTaps = new HashSet<>();
      final Scheme sourceScheme = new
AvroScheme(SCHEMA$);

      for (int argIndex = 0; argIndex <= args.length - 1;
argIndex++)
      {

         sourceTaps.add(new Hfs(sourceScheme, "s3n://" +
args[argIndex]));
      }

      Tap[] taps = sourceTaps.toArray(new
Tap[sourceTaps.size()]);
      MultiSourceTap inTap = new MultiSourceTap(taps);

      //preparing Sink tap
      String bucketPathWithS3Link = "s3n://" +  CountPath;
      Tap outTap = new Hfs(new TextDelimited(Fields.ALL,
","), bucketPathWithS3Link, SinkMode.REPLACE);

     // aggregating   hits
      Pipe assembly = new Pipe("Assembler");
      Pipe cntPipe = new Pipe("cntPipe", assembly);
      Fields selector = new Fields(COUNTFIELD);
      Fields cnt = new Fields("cnt");
      cntPipe = new CountBy(cntPipe, selector, cnt);

      //Initializing flowconnector
 FlowConnector flowconnector = new
HadoopFlowConnector(getPropertiesForCascadingJob());

      //connecting & executing the flow
      FlowDef flowDef = FlowDef.flowDef()
            .addSource(assembly, inTap)
            .addTailSink(cntPipe, outTap);

      flowconnector.connect(flowDef).complete();
  }
```

**Appendix 3: Cascading Count Unique jobs**

```
{ // Count unique
    // Source tap
      Set<Tap> sourceTaps = new HashSet<>();
      final Scheme sourceScheme = new
AvroScheme(SCHEMA$);
      for (int argIndex = 0; argIndex <= args.length - 1;
argIndex++)
      {

         sourceTaps.add(new Hfs(sourceScheme, "s3n://" +
args[argIndex]));
      }

      // Sink tap
      Tap[] taps = sourceTaps.toArray(new
Tap[sourceTaps.size()]);
      MultiSourceTap inTap = new MultiSourceTap(taps);
      //preparing Sink tap
      String bucketPathWithS3Link = "s3n://" +
UniqueUserPath;
      Tap outTap = new Hfs(new TextDelimited(Fields.ALL,
","), bucketPathWithS3Link, SinkMode.REPLACE);

       // Computing unique user
      Pipe assembly = new Pipe("Assembler");
      Pipe ccntPipe = new Pipe("ccntPipe", assembly);
      Fields selector = new Fields(UNIQUE_FIELD);
      ccntPipe = new Unique(ccntPipe, selector);

      FlowConnector flowconnector = new
HadoopFlowConnector(getPropertiesForCascadingJob());
      FlowDef flowDef = FlowDef.flowDef()
            .addSource(assembly, inTap)
            .addTailSink(ccntPipe, outTap);
      flowconnector.connect(flowDef).complete();
   }
```