10-30-2012

PUMA: Purdue MapReduce Benchmarks Suite

Faraz Ahmad
School of Electrical and Computer Engineering, Purdue University, faraz@purdue.edu

Seyong Lee
Oak Ridge National Labs

Mithuna Thottethodi
School of Electrical and Computer Engineering, Purdue University, muthuna@purdue.edu

T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University, vijay@purdue.edu

PUMA: Purdue MapReduce Benchmarks Suite

Faraz Ahmad

Seyong Lee

Mithuna Thottethodi

T. N. Vijaykumar

TR-ECE-12-11

October 30, 2012

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

PUMA: Purdue MapReduce Benchmarks Suite

Faraz Ahmad[†], Seyong Lee[‡], Mithuna Thottethodi[†], T. N. Vijaykumar[†]
[†]School of Electrical and Computer Engineering, Purdue University, IN, USA
[‡]Oak Ridge National Labs, Oak Ridge, TN, 37831
{faraz, mithuna, vijay}@purdue.edu lees2@ornl.gov

PUMA Suite

MapReduce[5] is a well-known programming model, developed within Google, for processing large amounts of raw data such as crawled documents or web request logs on a cluster of commodity hardware comprising of thousands of machines. MapReduce provides automatic data management and fault tolerance to improve programmability of clusters. In the MapReduce programming model, programmers specify a *Map* function which processes input data to generate intermediate data in the form of *<key, value>* tuples, and a *Reduce* function which further processes values associated with a particular key. Hadoop is an open-source implementation of MapReduce which is being improved and developed regularly by software developers / researchers and is maintained by Apache Software Foundation. Despite being vast efforts on the development of Hadoop MapReduce, there has not been a very rigorous work done on the benchmarks side.

During our work on MaRCO[2], we developed a benchmark suite[8], called "PUMA" which represents a broad range of MapReduce applications exhibiting application characteristics with high/low computation and high/low shuffle volumes. There are a total of 13 benchmarks, out of which Tera-Sort, Word-Count, and Grep are from Hadoop distribution. The rest of the benchmarks were developed in-house and are currently not part of the Hadoop distribution. The three benchmarks from Hadoop distribution are also slightly modified to take number of reduce tasks as input from the user as well as to generate final time completion statistics of jobs.

Benchmark details

The details of benchmarks, including command line execution format and input data set description can be found below.

[‡]Work was done while at Purdue.

1 Term-vector

determines the most frequent words in a host and is useful in analyses of a host's relevance to a search. Map emits <*host*,*termvector*> tuples where *termvector* is itself a tuple of the form <*word*, *1*>. Reduce discards the words whose frequency is below some cut-off, sorts the rest of the list per key in a descending order with respect to count and emits tuples of the form <*host*, *list(termvector)*>.

Input format: any document (usually a web document in text/xml format)

Output format: <*host*> <*termvector*>

Dataset: web documents [4] downloaded from wikipedia database[3]. Due to HDFS (Hadoop File System) limitations, the datasets needed some processing such as (i) copying all files from multiple hierarchical directories to one directory, (ii) merging multiple files together to create small number of large-sized files rather than large number of small-sized files, and (iii) eliminating special character file names.

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar termvectorperhost -m <num-
maps> -r <num-reduces> <input-dir> <output-dir>
```

2 Inverted-index

takes a list of documents as input and generates word-to-document indexing. Map emits <*word*, *docId*> tuples with each word emitted once per *docId*. Reduce combines all tuples on key <*word*> and emits <*word*,*list(docId)*> tuples after removing duplicates.

Input format: any document (usually a web document in text/xml format)

Output format: <*word*> <*docId*>

Dataset: web documents[4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar invertedindex -m <num-maps>
-r <num-reduces> <input-dir> <output-dir>
```

3 Self-join

is similar to the candidate generation part of the *a priori* data mining algorithm [1] to generate association among *k+1* fields given the set of *k*-field associations. Map receives *k*-sized candidate lists of the form {$element_1$, $element_2$, ..., $element_k$} in alphanumerically sorted order. Map breaks the lists into <{$element_1$, $element_2$, ..., $element_{k-1}$}, {$element_k$}> tuples. Reduce prepares a sorted list of all the map values for a given key by building <*{element*$_1$*, element*$_2$*, ..., element*$_{k-1}$}, {*ele-ment'*$_1$*, element'*$_2$*, ..., element'*$_j$}> tuples. From these tuples, *k+1-sized* candidates can be obtained by appending consecutive pairs of map values *element'*$_i$*, element'*$_{i+1}$ to the *k-1-sized* key. By avoiding repetition of *k-1-sized* key values for every pair of values in the list, the tuples are a compact representation of the *k+1-sized* candidates set.

Input format: {e_1,e_2, ..., e_k}

Output format: <e_1,e_2, ... e_{k-1}>< e_k, e_{k+1} >

Dataset: Synthetic data [4]

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar selfjoin -m <num-maps> -r
<num-reduces> <input-dir> <output-dir>
```

4 Adjacency-list

is similar to search-engine computation to generate the adjacency and reverse adjacency lists of nodes of a graph for use by PageRank-like algorithms. Map receives as inputs graph edges <*p,q*> of a directed graph that follows the power law of the World-wide Web. For the input, we assume the probability, that a node has an out-degree of *i*, is proportional to 1/(i^{skew}) with an average out-degree of 7.2. Map emits tuples of the form <*q, from_list{p}:to_list{}*> and <*p, from_list{}:to_list{q}*>. For a given key, reduce generates unions of the respective lists in the *from_list* and *to_list* fields, sorts the items within the union lists, and emits <*x, from_list{sorted union of all individual from_list}:to_list{sorted union of all individual to_list}*> tuples.

Input format: {*p*,*q*}

Output format: <*p*><*from{list_of_in_degree}:to{list_of_out_degree}*>

,<q><*from{list_of_in_degree}:to{list_of_out_degree}*>

Dataset: Synthetic data [4]

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar adjlist -m <num-maps> -r
<num-reduces> <input-dir> <output-dir>
```

5 k-means

is a popular data mining algorithm to cluster input data into *k* clusters [7]. k-means iterates to successively improve the clustering. We classify movies based on their ratings using movies rating data which is of the form <*movie_id, list{rater_id, rating}*>. We use random starting values for the cluster centroids. Map computes the cosine-vector similarity of a given movie with the centroids, and determines the centroid to which the movie is closest (i.e., the cluster to which it belongs). Map emits <*centroid_id, (similarity_value, movie_data)*> where *movie_data* is (*movie_id, list{rater_id, rating})*. Reduce determines the new centroids by computing the average of similarity of all the movies in a cluster. The movie closest to the average is the new centroid and reduce emits the new centroid's and all movies' tuples to be used in the next iteration. The algorithm iterates until the change in the centroids is below a threshold.

Input Format: {*movie_id*: *userid1_rating1, userid2_rating2, ...*}

Output Format: kmeans produces two types of outputs:

(a) <*centroid_num*><{*movie_id: userid1_rating1, userid2_rating2, ...*}> (list of all movies associated with a particular centroid)

(b) <*centroid_num*>< {*similarity_value*} {*centroid_movie_id*} {*num_members*} {*userid1_rating1, userid2_rating2, ...*}> (*new centroid*}

Datasets: movie ratings dataset[4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar kmeans -m <num-maps> -r
<num-reduces> <input-dir> <output-dir>
```

6 Classification

classifies the input into one of *k* pre-determined clusters (unlike *k-means*, the cluster centroids are fixed). Similar to *k-means*, *classification* uses movie rating data which is of the form <*movie_id, list{rater_id, rating}*>. Similar to *k-means*, Map computes the cosine vector similarity of a given movie with the centroids, and determines the centroid to which the movie is closest (i.e., the cluster to which it belongs). Map emits <*centroid_id, movie_id*>. Unlike *k-means*, the details of movie ratings are not emitted because there are no further iterations which may need the details. Reduce collects all the movies in a cluster and emits <*centroid_id, movie_id*>.

Input Format: {*movie_id: userid1_rating1, userid2_rating2, ...*}

Output Format: <*centroid_num*><*movieid*>

Datasets: movie ratings dataset[4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar classification -m <num-
maps> -r <num-reduces> <input-dir> <output-dir>
```

7 Histogram-movies

generates a histogram of input data and is a generic tool used in many data analyses. We use the movie rating data. Based on the average ratings of movies (ratings range from 1 to 5) we bin the movies into 8 bins each with a range of 0.5. The input is of the form *<rater_id, rating, date>* and the filename is *movie_id*. Map computes the average rating for a movie, determines the bin, and emits *<bin, 1>* tuples. Reduce collects all the tuples for a bin and outputs a *<bin, n>* tuple.

Input Format: {*movie_id: userid1_rating1, userid2_rating2, ...*}

Output Format: <*bin_value*><*num_of_movies*>

Datasets: movie ratings dataset [4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar histogram_movies ?m <num-
maps> -r <num-reduces> <input-dir> <output-dir>
```

8 Histogram-ratings

generates a histogram of the ratings as opposed to that of the movies based on their average ratings. The input is same as that for *histogram-movies*. Here, we bin the ratings of 1-5 into 5 bins and map emits <*rating, 1*> tuple for each review. Reduce collects all the tuples for a rating and emits a <*rating, n*> tuple.

Input Format: {*movie_id: userid1_rating1, userid2_rating2, ...*}

Output Format: <*rating* ><*num_of_user_reviews*>

Datasets: movie ratings dataset [4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar histogram_ratings ?m <num-
maps> -r <num-reduces> <input-dir> <output-dir>
```

9 Sequence-Count

generates a count of all unique sets of three consecutive words per document in the input data. Map emits <*word1|word2|word3|filename, 1*> tuples. Reduce adds up the counts for the multi-words from all map tasks and outputs the final count.

Input format: any document (usually a web document in text/xml format)

Output format: <*word1|word2|word3|filename*> <*count*>

Dataset: web documents [4].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar sequencecount -m <num-maps>
-r <num-reduces> <input-dir> <output-dir>
```

10 Ranked Inverted Index

takes list of words and their frequencies per document and generates lists of documents containing the given words in decreasing order of frequency. Map takes sequence-count benchmark's output <*word-sequence|filename,n*> as its input and separates counts from the rest of the data in the input. Map output format is <*word-sequence, {filename,n}*>. Reduce takes all map outputs and produces a list per word-sequence in decreasing order of occurrence in the respective documents <*word-sequence*><{*count1, file1*},{*count2, file2*}, ..>.

Input format: <*word-sequence|filename*><*count*>

Output format: <*word-sequence*> <*count | file*>

Dataset: Output of *Sequence-Count*.

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar rankedinvertedindex -m
<num-maps> -r <num-reduces> <input-dir> <output-dir>
```

11 Tera-sort

sorts 100-byte <*key,value*> tuples on the keys where key is a 10-byte field and the rest of the bytes as value (payload). Map is identity function which simply reads and emits the tuples and Reduce emits the sorted data to the final output. The sorting occurs in MapReduce's in-built sort while reduce tasks simply emit the sorted tokens.

Input format: {10-bytes key}{90-bytes value}

Output format: <10-bytes key><90-bytes value>

Dataset: Generated through *TeraGen* in *Hadoop* [6].

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar terasort <input-dir>
<output-dir> <num-reduces>
```

12 Grep

searches for a pattern in a file and is a generic search tool used in many data analyses. Map outputs lines containing either of the pattern as <*regex, 1*> tuples. Reduce task adds up the counts and emits <regex, n> tuples.

Input format: any document (usually a web document in text/xml format)

Output format: <regex> <count>

Dataset: web documents [4]

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar grep <input-dir> <output-
dir> <num-reduces> <regex> [<group>]
```

13 Word-count

counts the occurrences of each word in a large collection of documents. Map emits <word,1> tuples. Reduce adds up the counts for a given word from all map tasks and outputs the final count.

Input format: any document (usually a web document in text/xml format)

Output format: <word> <count>

Dataset: web documents [4]

Command-line execution:

```
$ bin/hadoop jar hadoop-*-examples.jar wordcount -r <num-reduces>
<input-dir> <output-dir>
```

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of 20th Intl. Conference on Very Large Data Bases, VLDB*, pages 487–499, 1994.

[2] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. MapReduce with Communication Overlap (MaRCO). In *Technical Report, ECE, Purdue University*, 2007.

[3] Wikipedia HTML data dumps. http://dumps.wikimedia.org/enwiki/.

[4] PUMA Datasets. http://web.ics.purdue.edu/~fahmad/benchmarks/datasets.htm.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[6] Hadoop. http://lucene.apache.org/hadoop/.

[7] J.Hartigan. *Clustering Algorithms. Wiley*, 1975.

[8] PUMA Benchmarks. http://web.ics.purdue.edu/~fahmad/benchmarks.htm