Delft University of Technology

Faculty Electrical Engineering, Mathematics and Computer Science

Large-Scale Data Management with HADOOP (IN4331)

Bouke Nederstigt	4008812
Anemona Voinea	4317602
Abhishek Sen	4319850

Contents

1 Introduction . . . 2
  1.1 Software and Tools . . . 2
2 Combiner Functions . . . 3
3 Movies . . . 4
  3.1 Major Components . . . 4
  3.2 Execution Details and Sample Output . . . 4
4 PigLatin Queries . . . 5
  4.1 Queries . . . 5
5 Inverted File Project . . . 6
  5.1 Major Components . . . 6
  5.2 Application Sequence . . . 6
  5.3 Sample Output . . . 7
  5.4 Analysis . . . 7
6 Graph Analysis . . . 8
  6.1 Counting Triangles in Giraph . . . 8
  6.2 Counting Triangles Via 3-way Join in MapReduce . . . 9
  6.3 Testing . . . 10
  6.4 Analysis . . . 10

1 Introduction

For this project we chose to investigate the use of HADOOP as a tool for large scale data management. This report gives an overview of the different experiments that we ran and their results. Some code and query excerpts will be shown in the report but the entire source code will be available on our GitHub repository. The repository also contains a README file that contains source code installation instructions for each of the applications.

1.1 Software and Tools

We faced some difficulties with software versions and compiling code as a result of incompatible packages. The table below lists the main software components that were used during this project.

Table 1: Tools used

Language	Java, Hadoop Java libraries, PigLatin
Development Environment	IntelliJ IDEA, Sublime Text 2
Operating System	Mac OSX, Ubuntu

2 Combiner Functions

The goal of this exercise was to practice building combiner functions. The initial MapReduce job from the book is adjusted to support the use of a combiner function. This means the application is a command-line tool which takes two arguments - The input document and output folder. To parse the authors files the application consists of the following components:

- com.hadoop.combiner.Authors - This class contains the Mapper, Combiner and Reducer subclasses for the Job. The mapper is the same as in the MapReduce job from the book and parses the file. The Combiner function in this application is the same as the reducer from the book. Because the Reducer does not add any functionality it is only used to write to the file. Normally the reducer would also contain the content from the combiner because one can never be certain that the combiner is executed. But for the sake of simplicity and overview we have left it out in this case.
- com.hadoop.combiner.AuthorsJob - This class contains the Job configuration. It is exactly the same as the Job file from the book, except that it also registers a combiner class.

As expected, the output from the application when using a combiner is exactly the same as when a reducer is used. An extract from the file is shown below:

Craig W. Thompson 1
D. Jason Penney 1
Daniel H. Fishman 2

3 Movies

This MapReduce Job takes XML files representing movies as input. Using a SAXParser the files are analyzed and split into two different files. One file containing lines with the title, actor names, year of birth and role. The second file contains the director's name, movie title and the year the movie was released.

3.1 Major Components

Listed below are the major components for this application:

- XMLInputFormat - To be able to parse XML files a MapReduce InputFormat is needed. For this the XMLINputFormat class from the Apache mahout project is used. This class enables the creation of a RecordReader and returns anything between specified XML tags. For this application we are only interested in content in between the 'movie' tags.
- ReadMovieXML - This SAXParser implementation does most of the heavy lifting and parses the data taken from the XMLInputFormat . It returns a movie's title, director, year and actors.
- Movies - The movie components consist of a Mapper and Reducer subclasses. The mapper creates an input stream that is parsed by the XML parser. For each director a key value pair with key '1' for each actor in a movie a key value pair with key '2' is written to the job context. The values contain the data that need to be written to the file. Depending on the key the Reducer writes the values to the corresponding files using multiple outputs. If key is '1' the values are written to a file "director", and if the key is 2 the values are written to the file "title".
- MoviesJob - Job configuration class - Besides the normal configuration items like in the combiner this class also sets the start and end tags for the XMLInputFormat. It also defines the Multiple Outputs "director" and "title" which are used to write to multiple files.

3.2 Execution Details and Sample Output

The application is run from the command line. Two parameters need to be specified. The input document and the output folder. Once this is done two files will be generated. The director file contains the director with title and production year. An extract from the file is shown below:

Michael Mann Heat 1995
Clint Eastwood Unforgiven 1992
Sam Raimi Spider-Man 2002

The title file contains the actors with the movies they played in, their birth date and role. Shown below is the sample output for this application:

Unforgiven Morgan Freeman 1937
Unforgiven Clint Eastwood 1930
Match Point Jonathan Rhys Meyers 1977
Match Point Scarlett Johansson 1984
Lost in Translation Scarlett Johansson 1984
Lost in Translation Bill Murray 1950

4 PigLatin Queries

Using the textual outputs from section 3, namely *title-and-actor.txt* and *director-and-title.txt*, we used PigLatin queries to answer the questions from section 19.5.3 of the Web Data Management textbook. The output generated from these commands is available in the GitHub directory under the ”19.5.3PigLatin queries” folder.

4.1 Queries

Load title-and-actor.txt and group on the title. The actors should appear as a nested bag.

```
titlesActors = LOAD ’/Users/Abhi/Desktop/title-and-actor.txt’ AS (title:chararray, actor:chararray, birth:int, role:chararray);
groupTitle = GROUP titlesActors BY title;
titles = FOREACH groupTitle GENERATE group, (titlesActors.actor, titlesActors.role);
```

Load director-and-title.txt and group on the director name. Titles should appear as a nested bag.

```
namesDirectors = LOAD ’/Users/Abhi/Desktop/director-and-title.txt’ AS (director:chararray, title:chararray, year:int);
groupDirector = GROUP namesDirectors by director;
titles = FOREACH groupDirector GENERATE group, (namesDirectors.title, namesDirectors.year);
```

Apply the cogroup operator to associate a movie, its director and its actors from both sources.

```
titlesActors = LOAD ’/Users/Abhi/Desktop/title-and-actor.txt’ AS (title:chararray, actor:chararray, birth:int, role:chararray);
namesDirectors = LOAD ’/Users/Abhi/Desktop/director-and-title.txt’ AS (director:chararray, title:chararray, year:int);
groupTitle = GROUP titlesActors BY title;
titles = FOREACH groupTitle GENERATE group, (titlesActors.actor);
cogrouped = COGROUP titles BY group, namesDirectors by title;
```

Write a PIG program that retrieves the actors that are also director of some movie.

```
namesActors = LOAD ’/Users/Abhi/Desktop/title-and-actor.txt’ AS (title:chararray, actor:chararray, birth:int, role:chararray);
namesDirectors = LOAD ’/Users/Abhi/Desktop/director-and-title.txt’ AS (director:chararray, title:chararray, year:int);
groupActorMovies = GROUP namesActors BY actor;
actorTitles = FOREACH groupActorMovies GENERATE group, (namesActors.title);
actorAndDirector = JOIN namesActors BY actor, namesDirectors BY director;
result = COGROUP actorTitles BY group, actorAndDirector BY director;
```

Write a modified version that looks for artists that were both actors and director of a same movie.

```
namesActors = LOAD ’/Users/Abhi/Desktop/title-and-actor.txt’ AS (title:chararray, actor:chararray, role:chararray);
namesDirectors = LOAD ’/Users/Abhi/Desktop/director-and-title.txt’ AS (director:chararray, title:chararray, year:int);
actorAndDirector = JOIN namesActors BY actor, namesDirectors BY director;
```

5 Inverted File Project

The goal of this part of the project was to build an inverted file using a MapReduce job that computed the term-frequency (tf), inverse-document-frequency (idf) and the tf-idf parameters for words within a given text blob. The testing was done using the 'summary' sections from the *movies.xml* database that we have used from earlier assignments.

5.1 Major Components

The Java program is built as a command-line tool and takes 2 or more arguments. The first to second last arguments are the input files on which the MapReduce operations are performed. The input files need to be in XML format and should contain a 'summary' tag else the tf-idf calculations will be skipped. Listed below are the major components for this application:

- com.hadoop.movies.ReadMovieXML - This class parses the 'summary' tags from input XML files to produce a continous stream of tokenizable words.
- com.hadoop.movies.Movies - There are 3 subclasses in this package - MoviesMapper, MoviesCombiner and MoviesReducer. These classes are used to perform the map, combine and reduce functions.
- com.hadoop.tfidf.TfIdf - Computes tf, idf and tf-idf values

5.2 Application Sequence

Figures 1 and 2 show sequence diagrams of how the mapper, combiner and reducer functions work together to compute tf, idf and tf-idf values to and output them to a file.

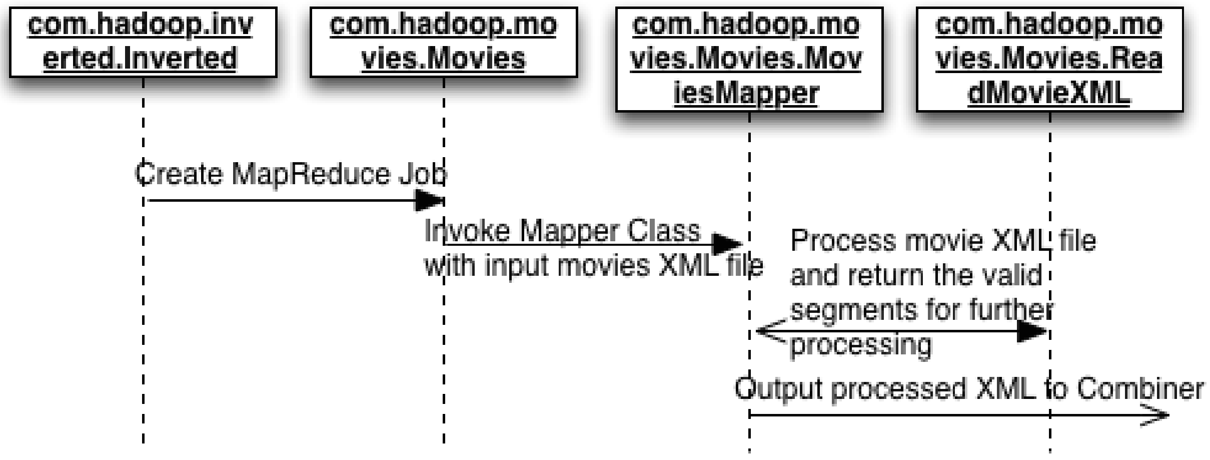

com.hadoop.inverted.Inverted
com.hadoop.movies.Movies
com.hadoop.movies.Movies.MoviesMapper
com.hadoop.movies.Movies.ReadMovieXML
Create MapReduce Job
Invoke Mapper Class with input movies XML file
Process movie XML file and return the valid segments for further processing
Output processed XML to Combiner

Figure 1: Inverted File Mapper Sequence

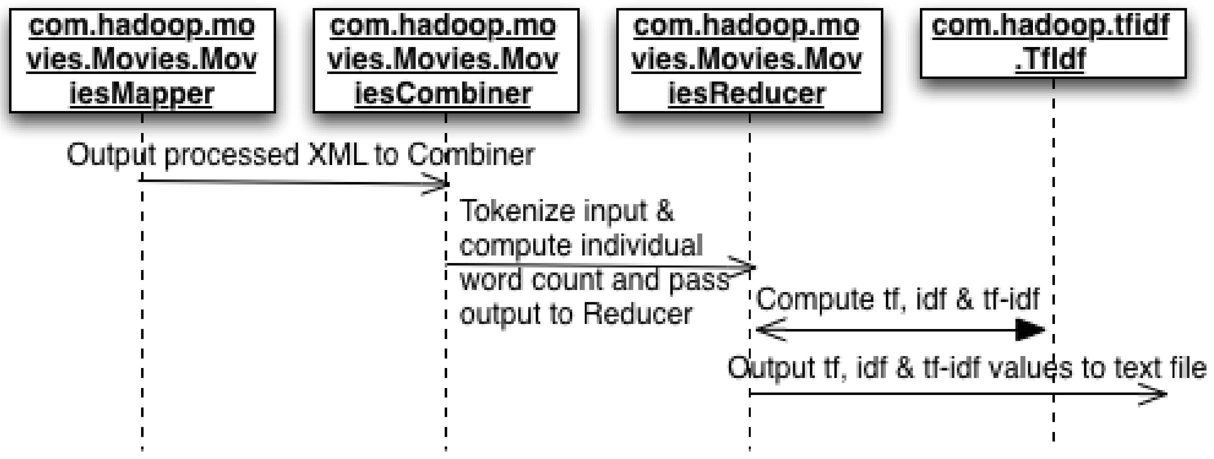
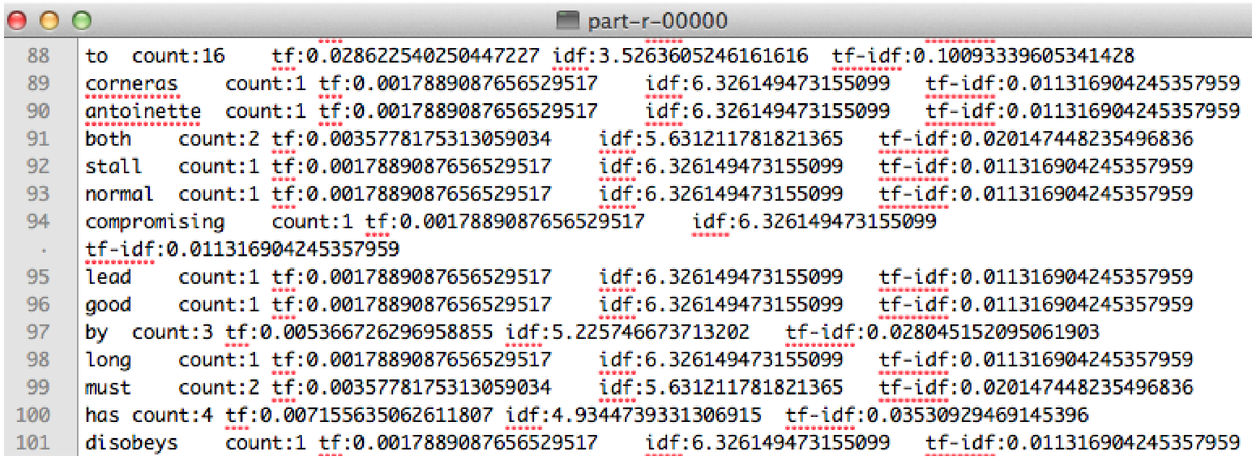

com.hadoop.movies.Movies.MoviesMapper
com.hadoop.movies.Movies.MoviesCombiner
com.hadoop.movies.Movies.MoviesReducer
com.hadoop.tfidf.TfIdf
Output processed XML to Combiner
Tokenize input & compute individual word count and pass output to Reducer
Compute tf, idf & tf-idf
Output tf, idf & tf-idf values to text file

Figure 2: Inverted File Mapper-Combiner Sequence

5.3 Sample Output

At the end of the tf-idf calculation process, a tab-delimited text file is created in the 'result' directory of the project that shows the word, the number of times it has occurred within the document (count) and the tf, idf and tf-idf values for that word in a tab-delimited file. Figure 3 below show a snippet of the sample output file generated using the *movies.xml* database as an input.



```
to	count:16	tf:0.028622540250447227	idf:3.5263605246161616	tf-idf:0.10093339605341428
corneras	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
antoinette	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
both	count:2	tf:0.0035778175313059034	idf:5.631211781821365	tf-idf:0.020147448235496836
stall	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
normal	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
compromising	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
lead	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
good	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
by	count:3	tf:0.005366726296958855	idf:5.225746673713202	tf-idf:0.028045152095061903
long	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
must	count:2	tf:0.0035778175313059034	idf:5.631211781821365	tf-idf:0.020147448235496836
has	count:4	tf:0.007155635062611807	idf:4.9344739331306915	tf-idf:0.03530929469145396
disobeys	count:1	tf:0.0017889087656529517	idf:6.326149473155099	tf-idf:0.011316904245357959
```

Figure 3: Snippet from IDF output document

5.4 Analysis

Future improvments to this application would be:

- Be able to accept any type of files (XML, text, etc.)
- Better tokenization of abbreviations (Col., Lt., Dr., etc.)

6 Graph Analysis

6.1 Counting Triangles in Giraph

a Overview of Implementation

The first implementation makes use of Giraph's message passing paradigm to send messages along a triangle's edges in order to discover it. The *TriangleCounterInGiraph* class extends Giraph's *BasicComputation* class to carry out a 4-step computation on the graph expressed with *LongLongNullTextInputFormat*. The final value of each vertex represents the number of triangles in which that vertex is the one with the smallest identifier and is written to the output with *IdWithValueTextOutputFormat*. Adding these values up gives the final number of triangles in the graph.

b Algorithm Description

In the first superstep, all vertices send their identifiers in messages to their neighbors which have a greater identifier than themselves. Thus, messages travel on edges of type XY (smallest vertex in triangle to middle vertex). Next, all vertices forward the messages they received to their neighbors which have an even greater identifier. Messages, now travel on edges of type YZ (middle vertex to greatest vertex in the triangle). The last edge is discovered when all vertices forward the messages they received to all their neighbors. When each vertex counts how many messages received in the previous step contain their identifier, they obtain the number of triangle in which they have the smallest identifier.

The pseudocode of the implementation is given below:

```
//Step 1
    foreach neighbor in Neighbors
        if neighbor > self
            send self.id to neighbor
//Step 2
    foreach message in ReceivedMessages
        foreach neighbor in Neighbors
            if neighbor > self
                send message to neighbor
//Step 3
    foreach message in ReceivedMessages
        foreach neighbor in Neighbors
            send message to neighbor
//Step 4
    count = 0
    foreach message in ReceivedMessages
        if message == self.id
            count++
    self.value = count
```

6.2 Counting Triangles Via 3-way Join in MapReduce

a **Overview of Implementation**

The second implementation attempts a 3-way join between 3 copies of the edge relation. *TriangleCounterInHadoop* uses the Mapper-Reducer combination to first send edges to their appropriate reducers in the theoretical "matrix" and then compute at each reducer how many triangles are formed. By default, the number of buckets (b) used is 2, resulting in $2^3 = 8$ reducers needed to count triangles. This value can be overridden via a third command-line option (after the basic input and output path parameters). As in the previous case, the output is a set of pairs (reducer_number reducer_count). Summing up all the reducers' counts gives the final number of triangles.

b **Algorithm Description**

The algorithm follows that described in the "Optimizing Multi-Way Join on MapReduce" lecture given by dr. Jacek Sroka. The mapper reads the input and creates only edges (a, b) with a ¡ b. It computes h(a) and h(b) and sends edge (a,b) to reducers of the form:

(a) (h(a), h(b), *), as an XY edge (from smallest vertex in triangle to the middle vertex)

(b) (*, h(a), h(b)), as an YZ edge (from middle vertex to the greatest vertex in the triangle)

(c) (h(a), *, h(b)), as an XZ edge (from the smallest to the greatest vertex in the triangle)

The reducers each receive edges of the 3 different types. For each edge of type XY, they search for a compatible YZ edge and, if the closing XZ edge is also found, they count that triangle once.

The pseudocode of the implementation is given below:

```
//Mapper
    set b to number of buckets
    foreach edge (v1, v2) in Input
        if v1 < v2
            h1 = modulo(v1, b)
            h2 = modulo(v2, b)
            foreach i < b
                send (v1, v2, XY) to reducer h1*b^2 + h2*b + i
                send (v1, v2, YZ) to reducer i*b^2 + h1*b + h2
                send (v1, v2, XZ) to reducer h1*b^2 + i*b + h2

//Reducer
    count = 0
    foreach (v1, v2, XY) in Received
        foreach (v2, v3, YZ) in Received
            if Received contains (v1, v3, XZ)
                count++
    self.output = count
```

6.3 Testing

Automated tests were carried out to compare the results of the two implementations with the result of the trivial algorithm computed on the same input. The pseudocode of this algorithm is given below:

```
foreach (v1, v2) in Edges
    foreach (v2, v3) in Edges
        if Edges contains (v1, v3)
            count++
```

The graph for each test is generated by randomly adding edges to a set of vertices until reaching a specific density of edges, between 0 (no edges) and 1 (complete graph). The tests are automatic and can be run any time using the TriangleCountingTester. We carried out 15 tests, involving graphs with 10, 20, 30, 50, and 100 vertices and densities of 0.4, 0.6, and 0.8 (these can be found in the project folder). The three resulting values were identical in all cases.

6.4 Analysis

It is worth mentioning that the cost in communication or runtime was not evaluated in the tests.

Optimizations on the code are possible for both implementations. For instance, the Giraph version would benefit from a preliminary renaming phase of the nodes. Since messages are passed to neighbors with a greater identifier, it would be optimal to have the vertices with fewer neighbors (smaller degree) have smaller identifiers. Likewise, the multi-way join could suffer the modifications expressed in the book ”Mining of Massive Datasets” by Anand Rajaraman and Jeffrey Ullman (available at `http://infolab.stanford.edu/~ullman/mmds/book.pdf`). The authors suggest ”renaming” each vertex to the pair (h(v), v) and reordering the vertices. Thus, the number of reducers and, consequently, the communication cost, is cut down to a third, since only reducers (i,j,k), with i, j, k in ascending order, remain.

Moreover, the Hadoop jobs were run using the default configuration, in a pseudo-distributed manner. Greater speed and computation efficiency can be obtained by tailoring the runtime environment to the specific problem.