December 2015

Entity Resolution in Big Data

Duc Minh Pham
Worcester Polytechnic Institute

Thanh Long Vu
Worcester Polytechnic Institute

ELODU: ENTITY RESOLUTION IN BIG DATA

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Computer Science

Submitted by:

Duc M. Pham

Thanh Long X. Vu

December 18, 2015

APPROVED:

Advisor: Professor Mohamed Y. Eltabakh

1. ABSTRACT

Today, with the rapid development of technology, human entered a new era of Information Technology. Computer appears in every aspects of life. Data is being transfer from paper to digital. Therefore, the demand of data storage is increasing quickly. Human need a new technique to handle Big Data, that's why Hadoop was born. "The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models" – Apache [1]. However, the conflicts and duplicates of data is still happen in many cases. In this report, we will illustrate a new technique for entity resolution in big data. This technique is based on Hadoop, using one join target field along with the help of supporting fields, and a method for similarity join from University of California, Irvine called "Efficient Parallel Set-Similarity Joins Using MapReduce". [2]

TABLE OF CONTENTS

2. INTRODUCTION

2.1 OVERVIEW

Big Data is becoming a persistent problem in the field of Computer Science. With the rapid increase in demand of Data Storage everywhere, traditional data processing methods are starting to look inefficient. They are inadequate to handle Big Data. Data is expanding so fast and the amount of data increases rapidly means controlling is becoming harder. With old ways of handling data, operations on it cannot be done so quickly anymore. Every second that passes, there are billions of megabytes of data transferred through the Internet. Moreover, data formats are not standardized. Data formats differ from place to place. This leads to data conflict and duplications. For example, in Vietnam names are stored as Last Name first, then Middle Name and finally First Name, but many other countries in the world use different formats (for example First Name, Middle Name and then Last Name). Therefore, we need a new technique to handle this problem. Hadoop is created to help people storing and processing Big Data. This report will present a new technique based on Hadoop to process large datasets with the same entities but stored in different formats. We developed a custom input format to automatically correct the wrongly formatted attributes into the standard format. We tested our code by comparing the results given when the default TextInputFormat is used and when our CustomInputFormat is used. In the next part, we will talk about Goals and Objectives of this project.

2.2 GOALS AND OBJECTIVES

This project has six goals:

1. Creating Configuration File that gives information about the target field, supporting fields, matching type of these fields and the threshold for similarity join.
2. Writing java program to extract the data for testing purpose from two real big datasets: DBLP and CiteseerX.
3. Implementing "Efficient Parallel Set-Similarity Joins Using MapReduce" - University of California, Irvine for similarity joining method.
4. Enhancing our join results by removing duplications.
5. Developing a custom Input Format using the output we acquired to automatically format the wrong attributes.
6. Testing our input format by joining and doing an aggregation to group the records by target field and count the number of records grouped

And the most important goal is the 5th goal. In the end we want to create a new input format to replace the default TextInputFormat, which will be able to detect wrongly formatted attributes and fix them.

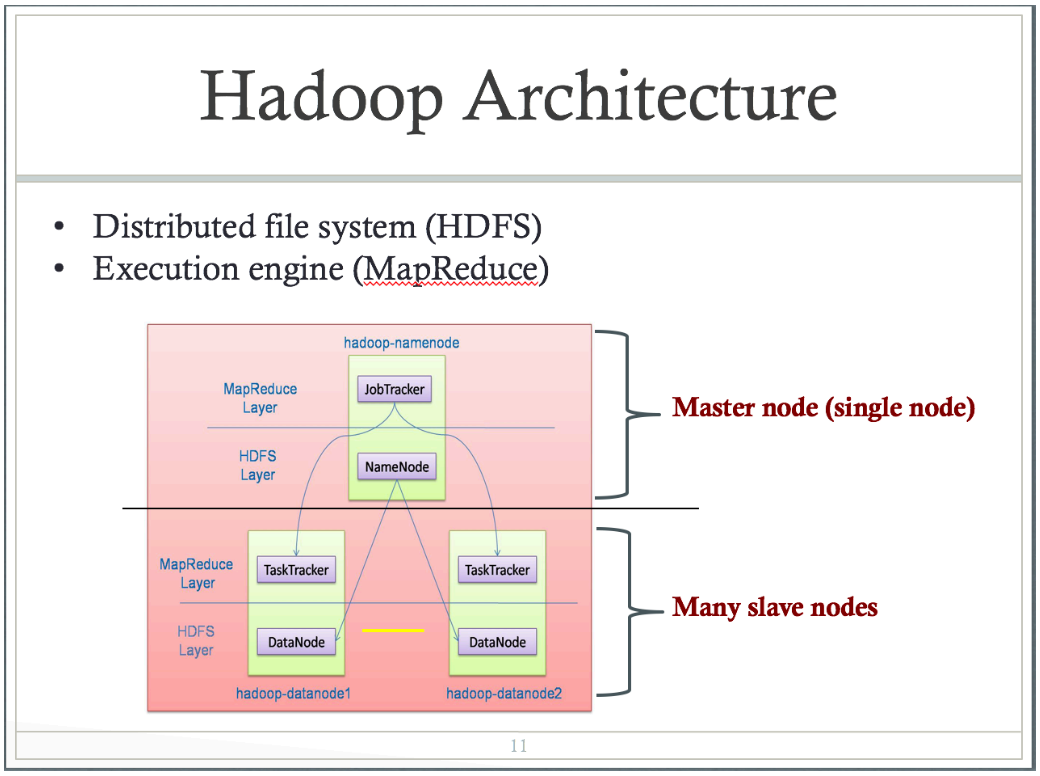
2.3 HADOOP

Hadoop is an open source software project from Apache that “allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures” [1]

Apache™ Hadoop® system consists of 4 modules:

1. The first part is Hadoop Common: “The common utilities that support the other Hadoop modules” [1]
2. The second part is Hadoop Distributed System: “A distributed file system that provides high-throughput access to application data” [1]
3. The third part is Hadoop YARN: “A framework for job scheduling and cluster resource management” [1]
4. The final part is Hadoop MapReduce: “A YARN-based system for parallel processing of large data sets” [1]

The graphs below will show the Architecture of Apache™ Hadoop®:



Hadoop Architecture

- Distributed file system (HDFS)
- Execution engine (MapReduce)

Figure 1: Hadoop Architecture

Hadoop uses *master-slave, shared-nothing* architecture. It has a Master node as a single node and many slave nodes. Each node will have main layers. The first layer is MapReduce functions layer. This layer will handle the methods used for processing data. MapReduce tasks will have JobTracker to manage nodes in the cluster. If the JobTracker is failed, all the jobs will be halted. The second main layer is the HDFS layer. HDFS layer helps storing large volumes of data. Data storage in HDFS is scalable and reliable.

The graph below illustrates how Apache™ Hadoop® Map-Reduce executes:

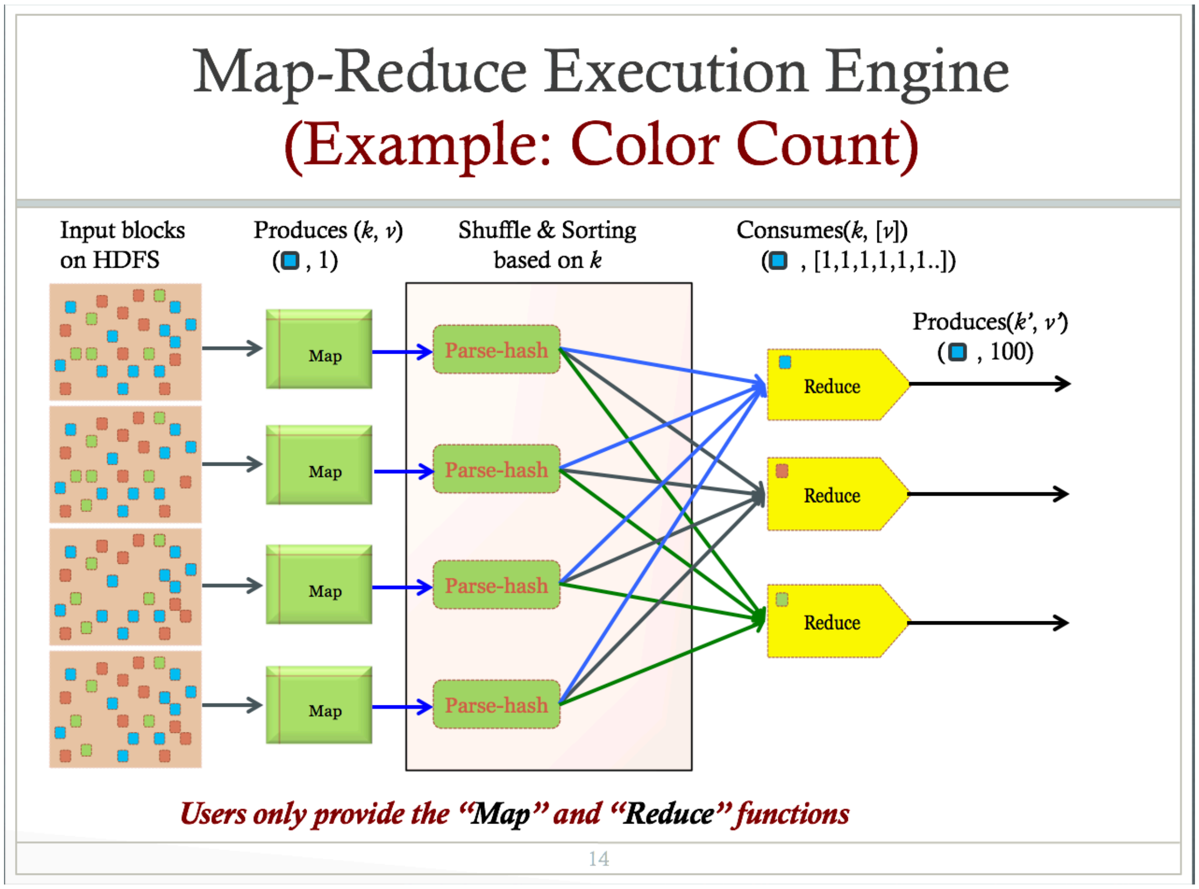

Map-Reduce Execution Engine
(Example: Color Count)
Input blocks on HDFS
Produces (k, v) (, 1)
Shuffle & Sorting based on k
Consumes(k, [v]) (, [1,1,1,1,1,1..])
Produces(k', v') (, 100)
Map
Parse-hash
Reduce
Users only provide the "Map" and "Reduce" functions
14

Figure 2: Color Count example

Data storage in HDFS will be divided into smaller chunks. They are usually separated by size. These data chunks will randomly go to Mappers, and from them The Mappers produce key and value pairs. The key-value pairs are then shuffled and sorted based on their keys. Data is then passed to reducers such that the pairs that have the same key will go to the same reducers. Reducers then combine, reduce data or do any kind of processing on it before the final result is given.

3. SYSTEM OVERVIEW

3.1 WORK FLOW

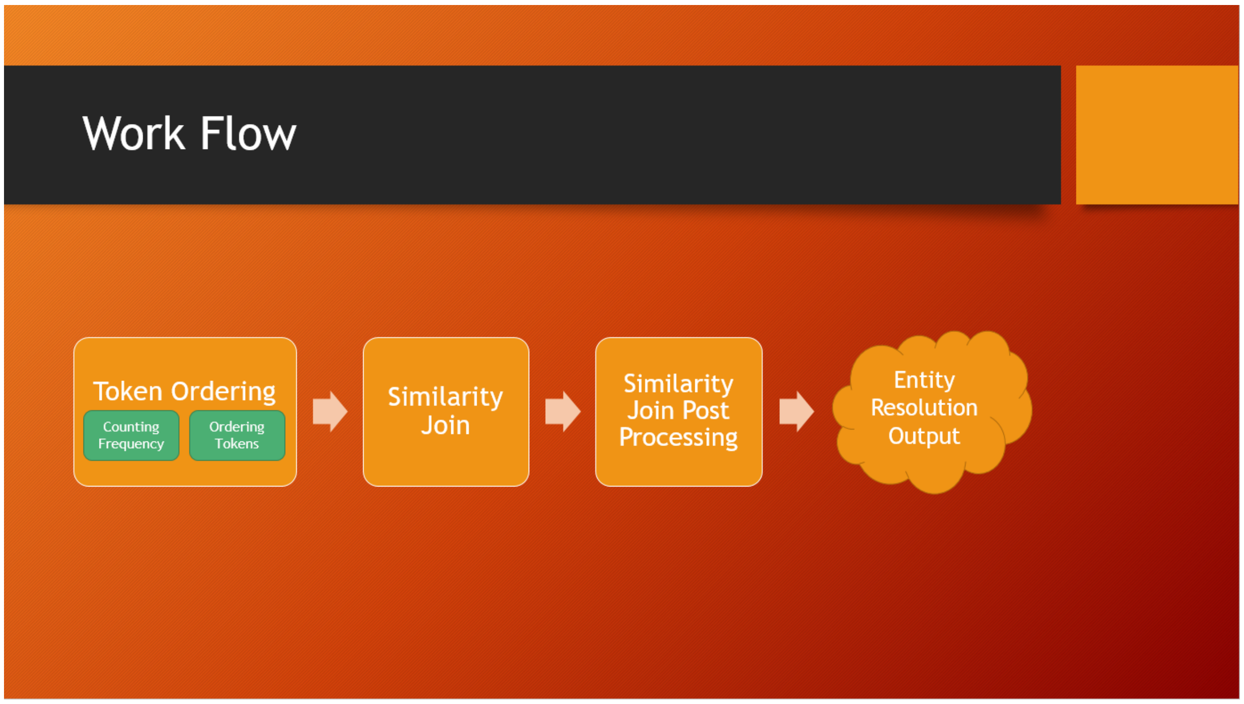


Figure 3: Work Flow of Project

Overall, to build our custom input format, we first use four main Map-Reduce jobs: Counting Frequency, Ordering Tokens, Similarity-Join and finally Similarity-Join Post Processing. Each of these jobs is a separate java class with different mappers and reducers. The first job (Counting Frequency) takes a data file stored in HDFS as input. It finds the join field in each record in the dataset, divides it into tokens, counts the frequency of each token and writes the tokens along with their frequency counts to output. The second job (Order Tokens) swaps key and value position and sorts the tokens. The third job (Similarity-Join) takes the two datasets and the token ordering as input. It uses the order of tokens to find the candidate records, then checks the candidate records to find the records that do join together. The fourth job (Similarity-Join Post Processing) filters the output from the previous job to get cleaner result. The return values of post processing job will then be used to make the custom input format.

3.2 USE CASES WORK FLOW

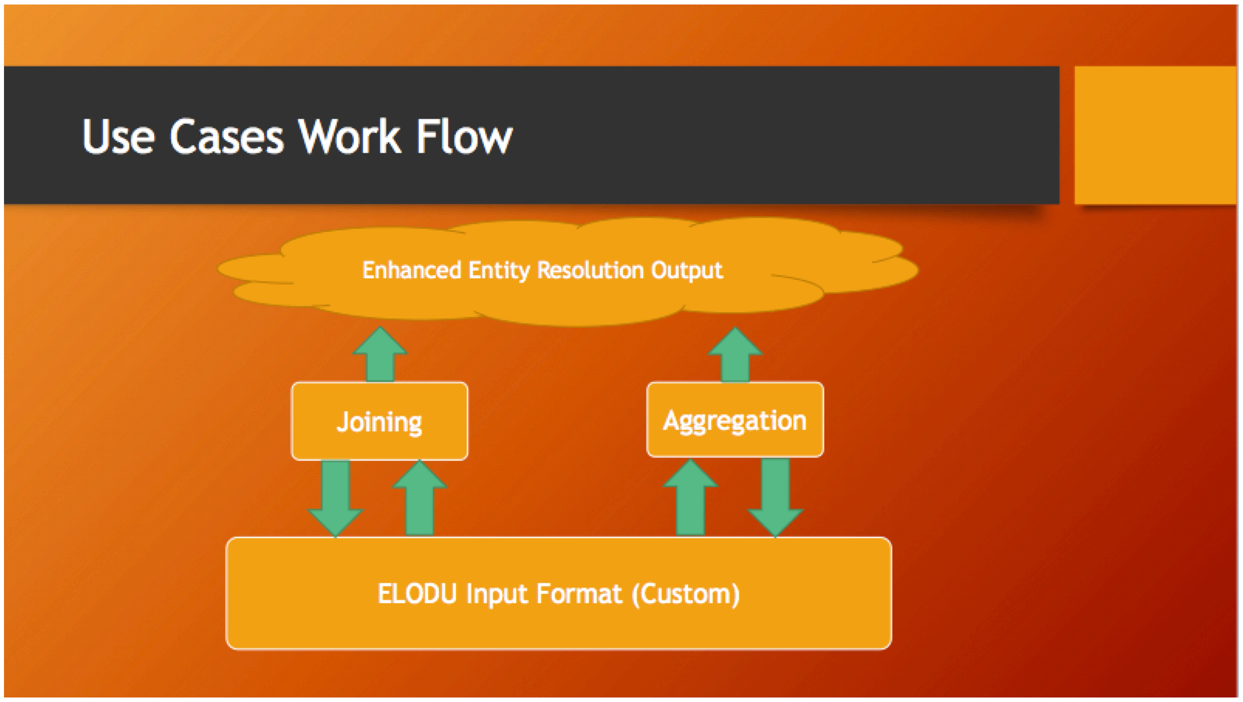

Use Cases Work Flow
Enhanced Entity Resolution Output
Joining
Aggregation
ELODU Input Format (Custom)

Figure 4: Diagram of Use Cases Work Flow

In this project, we will write two program with different use cases for testing and evaluation. The first use case is joining two datasets, and the second use case is doing aggregation, in which records are grouped by the target field and the records grouped together are counted. Both use cases will be processed once with the default Text Input Format and once with our custom ELODU Input Format to evaluate the efficiency and accuracy between the two. The enhanced entity resolution output is the final result of this project.

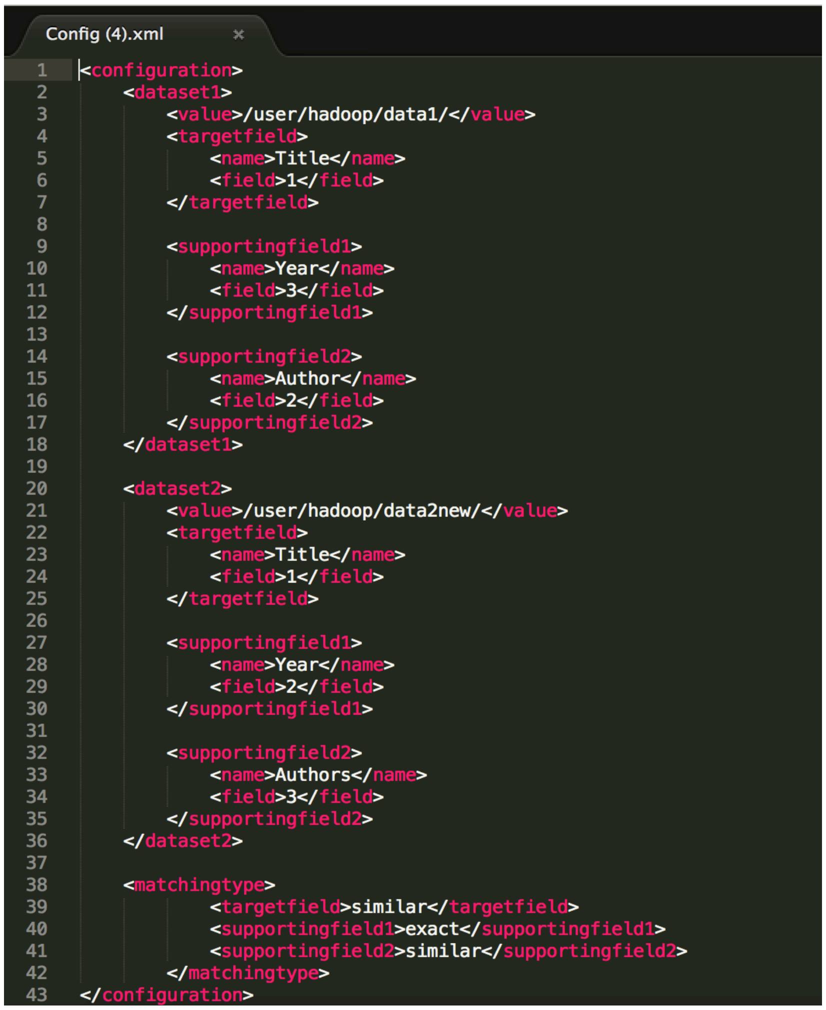
3.3 PROBLEMS, LIMITATION AND SOLUTIONS

Doing this project, we had to face many problems and limitations. The first challenge that we had to face is building the two datasets. At first we wanted to write a java program to build two big sample datasets by randomizing strings. This idea is feasible, but unreal and meaningless data did not seem very helpful in test cases. Therefore we decided to use real datasets and we found the datasets that were used in the "Efficient Parallel Set-Similarity Joins Using MapReduce" paper [2]. These datasets are called DBLP [3] and CiteseerX [4]. They consists of publications along with their author names, publish years, abstracts, links and so on. The DBLP dataset is 1.67 GB large and the CiteseerX dataset is 7.6 GB large. They are both in XML format, therefore we needed to write java programs to remove the tags and extract the useful parts inside the files. The DBLP data file is not hard to convert since the format is clear for the most part. However, the CiteseerX data file has many records with faulty format, so to serve our purpose the best we divided the file and extracted the parts that were in good form.

The second challenge for us is studying Hadoop. Since we did not know about Hadoop before starting the project, we needed quite some effort to research and study to understand how it works and to be able to write Map-Reduce jobs. We went through sample codes both from the Internet and inside the system, and since there are codes that use different versions of Hadoop and different Map-Reduce APIs everywhere, sometimes we found it a bit confusing choosing the correct path to follow.

The third problem that we faced was running out of disk space. The data files are quite large and we had to write a lot of draft lines to console when debugging the programs. This caused the Map-Reduce Administrator site to freeze or Terminal shell exploded occasionally.

3.4 ENTITY RESOLUTION CONFIGURATION



```xml
<configuration>
    <dataset1>
        <value>/user/hadoop/data1/</value>
        <targetfield>
            <name>Title</name>
            <field>1</field>
        </targetfield>

        <supportingfield1>
            <name>Year</name>
            <field>3</field>
        </supportingfield1>

        <supportingfield2>
            <name>Author</name>
            <field>2</field>
        </supportingfield2>
    </dataset1>

    <dataset2>
        <value>/user/hadoop/data2new/</value>
        <targetfield>
            <name>Title</name>
            <field>1</field>
        </targetfield>

        <supportingfield1>
            <name>Year</name>
            <field>2</field>
        </supportingfield1>

        <supportingfield2>
            <name>Authors</name>
            <field>3</field>
        </supportingfield2>
    </dataset2>

    <matchingtype>
            <targetfield>similar</targetfield>
            <supportingfield1>exact</supportingfield1>
            <supportingfield2>similar</supportingfield2>
        </matchingtype>
</configuration>
```

Figure 6: Configuration file structure

The image above shows the configuration file that we used throughout this project. Each field is stored along with its column number for easy access and manipulation of data. The matching type section specifies the type of matching for each field, for example the supporting field number 1's matching type is exact so anything not exactly the same is considered not a match.

3.4.1 TARGET FIELD

Target field is the first part in configuration file. The target field is to point out the field that the datasets will be joined on, and therefore is the field that is the most important and needed the most processing. The test datasets that we used consists of records about publications. There are many fields inside one record such as author, title, publisher and so on. The target field we use for testing is the article name. As stated above, the "field" attribute gives information about the target field's location (Column number).

3.4.2 SUPPORTING FIELDS

Supporting fields is the second part of the configuration file. In this project we have two supporting fields: the publish year and the author name. These fields help us identify the entities with better accuracy because of the additional checking done on them besides the target field check. The matching type for supporting field can be exact match or similarity match.

3.3.3 MATCHING TYPE

Matching type is the third part of configuration file. It enables our program to identify which kind of matching will be suitable for which field. This project will focus on two kinds of matching: similarity match and exact match. Exact match means the values for that field have to be exactly the same to match, while similarity match means the values for that field just need to be similar to a specific degree.

3.3.4 THRESHOLD

Threshold is the limit value that is used for similar match. The similarity of two values will be computed and compared against this threshold, and if it's larger than or equal to the threshold, the two values will be considered the same.

4. SYSTEM IMPLEMENTATION

4.1 TOKEN ORDERING - PART 1 - COUNTING FREQUENCY

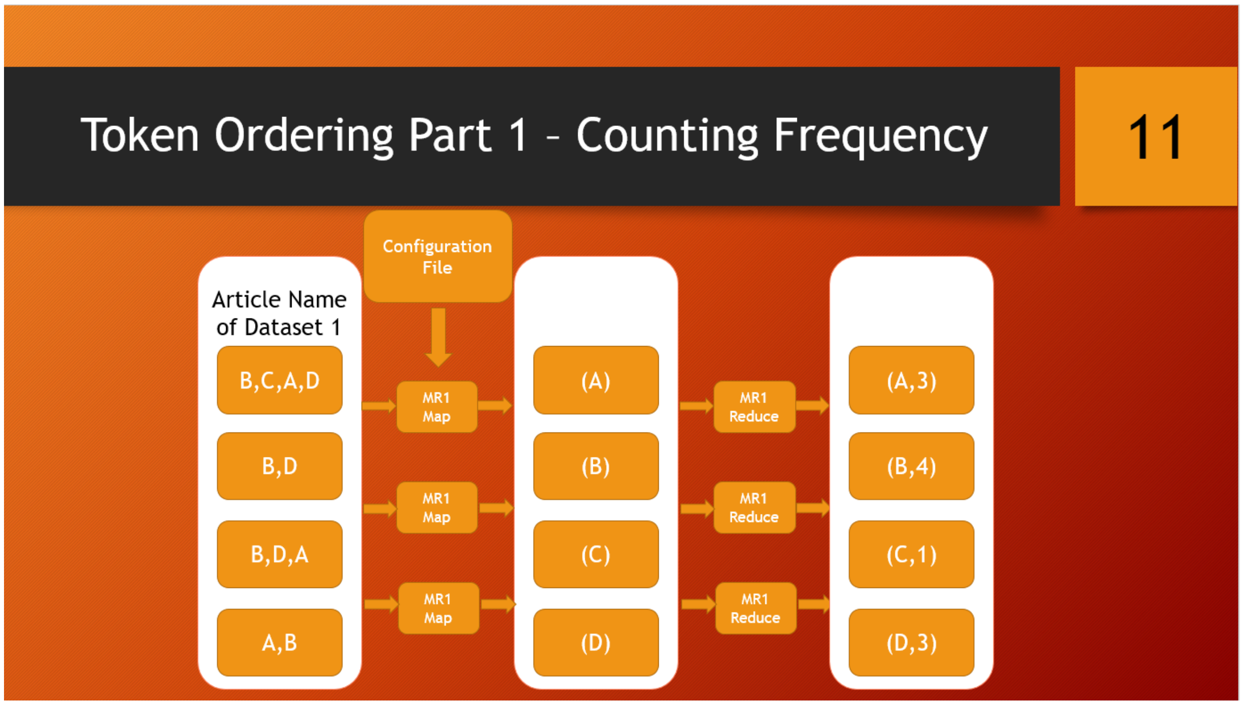


Figure 7: Data flow of token ordering part 1

The first MapReduce job counts the frequencies of each word that appears in the target field of the smaller dataset's records. We take the smaller dataset and the configuration file as input for the map phase. Using the information of the target field in the configuration file, the mappers extract only that field in each record, and separate it into single words (tokens). Each token will be a key, and the value for each key will be one. That way, all the occurrences of a word will be sent to the same reducer because key-value pairs are grouped by their keys before they are passed to the reduce phase. The reducers then add all the values to find the frequency of each token and write the result to the output file.

4.2 TOKEN ORDERING - PART 2 – ORDER TOKEN

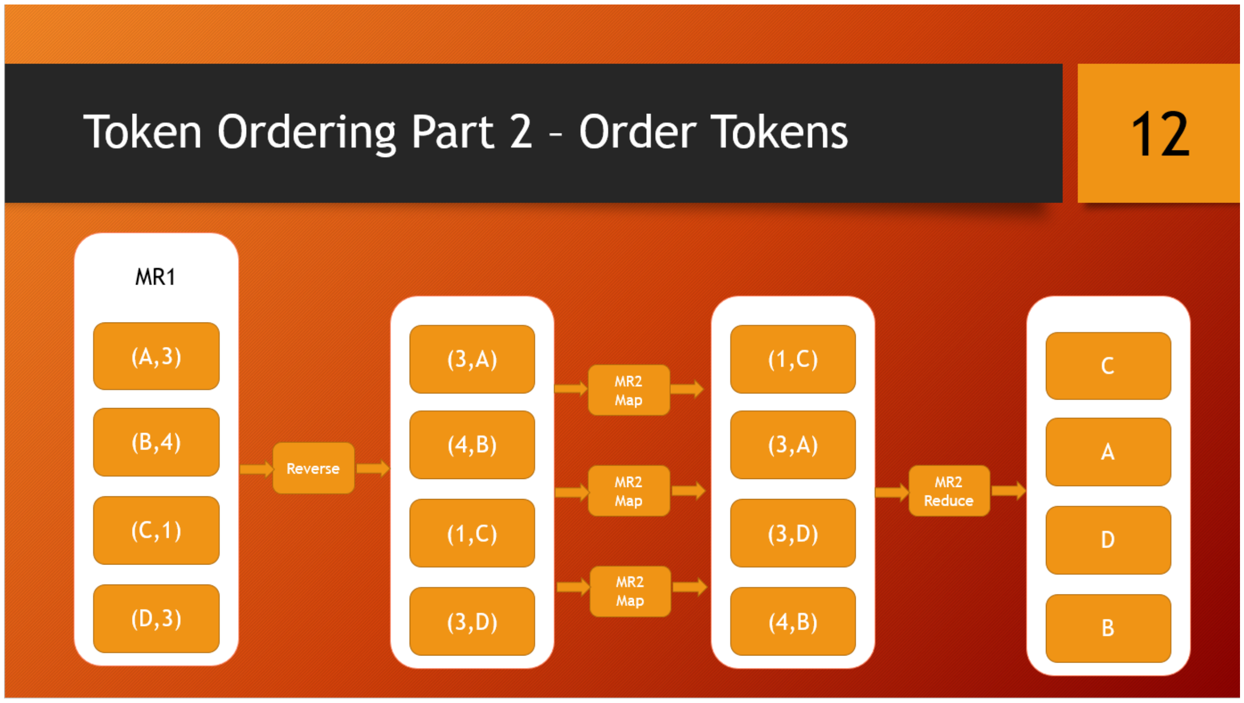


Figure 8: Data flow of token ordering part 2

The second MapReduce job sorts the tokens based on their frequencies. The mappers swap the key-value pairs received from the first job, so that the keys will be the frequency of the words and the values will be the words themselves. We use a single reducer for this job, because when the key-value pairs are passed to the reducer they will already be sorted on the frequency. The reducer then simply leaves out the frequency and outputs the words in ascending order of their counts.

4.3 SIMILARITY-JOIN

The first requirement for the join is the Configuration file. Configuration file stores information about the join target field, the supporting fields, the matching types for each field and the thresholds to use when the matching type is Similar Match. Moreover, it also provides the location for each field, which makes accessing specific parts inside the data so much easier. Another essential element of the join is the actual datasets. This project uses two big datasets that was generated from two real datasets (DBLP dataset [3] and CiteseerX [4]). We created them by writing a java program to extract the text data from XML format and remove unnecessary elements. With the data extracted, first our program will use the configuration file to identify the target field, supporting fields and matching type of each field. After that, the program will start checking the data based on all the acquired information. The result of the join depends on whether the fields match or not and how they match. The supporting fields give the join better accuracy so the join result is more correct. Finally, the result will be process through a filter to be cleaned up.

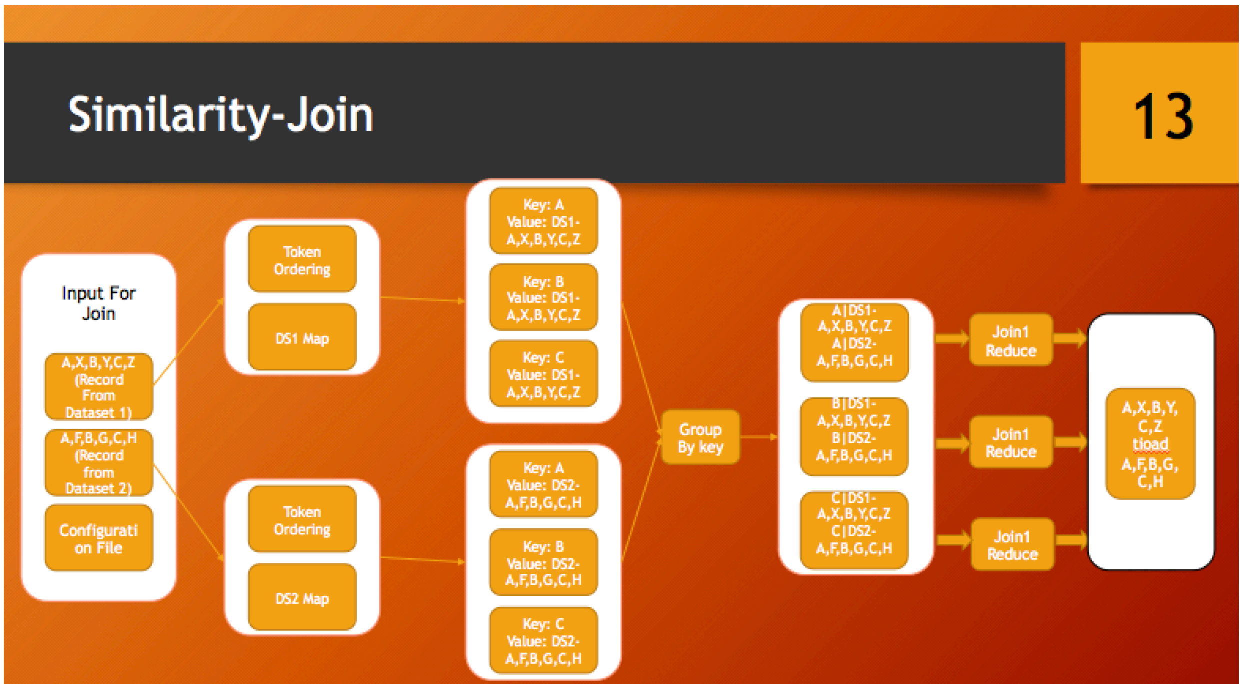

Similarity-Join
13
Input For Join
A,X,B,Y,C,Z (Record From Dataset 1)
A,F,B,G,C,H (Record from Dataset 2)
Configurati on File
Token Ordering
DS1 Map
Token Ordering
DS2 Map
Key: A Value: DS1-A,X,B,Y,C,Z
Key: B Value: DS1-A,X,B,Y,C,Z
Key: C Value: DS1-A,X,B,Y,C,Z
Key: A Value: DS2-A,F,B,G,C,H
Key: B Value: DS2-A,F,B,G,C,H
Key: C Value: DS2-A,F,B,G,C,H
Group By key
A|DS1-A,X,B,Y,C,Z A|DS2-A,F,B,G,C,H
B|DS1-A,X,B,Y,C,Z B|DS2-A,F,B,G,C,H
C|DS1-A,X,B,Y,C,Z C|DS2-A,F,B,G,C,H
Join1 Reduce
Join1 Reduce
Join1 Reduce
A,X,B,Y, C,Z tload A,F,B,G, C,H

Figure 9: Data flow of Similarity-Join

The third MapReduce job computes the similarity of the records and compare it with the thresholds given in the configuration file to decide which record in one dataset is actually which record in the other dataset. This job takes the tokens ordering, the two actual datasets and the configuration file as input. The map phase generates (token, record) pairs. Each record (as the value) will be tagged with the data file it comes from. The token (as the key) taken from the target field of each record is selected so that it has the smallest frequency count in the tokens ordering generated above. We take at most 3 tokens for each record. The reducers verify the records that have the same token using the supporting fields and the thresholds, and write the verified target fields R.a and S.a to the output file.

4.4 OUTPUT ENHANCEMENT

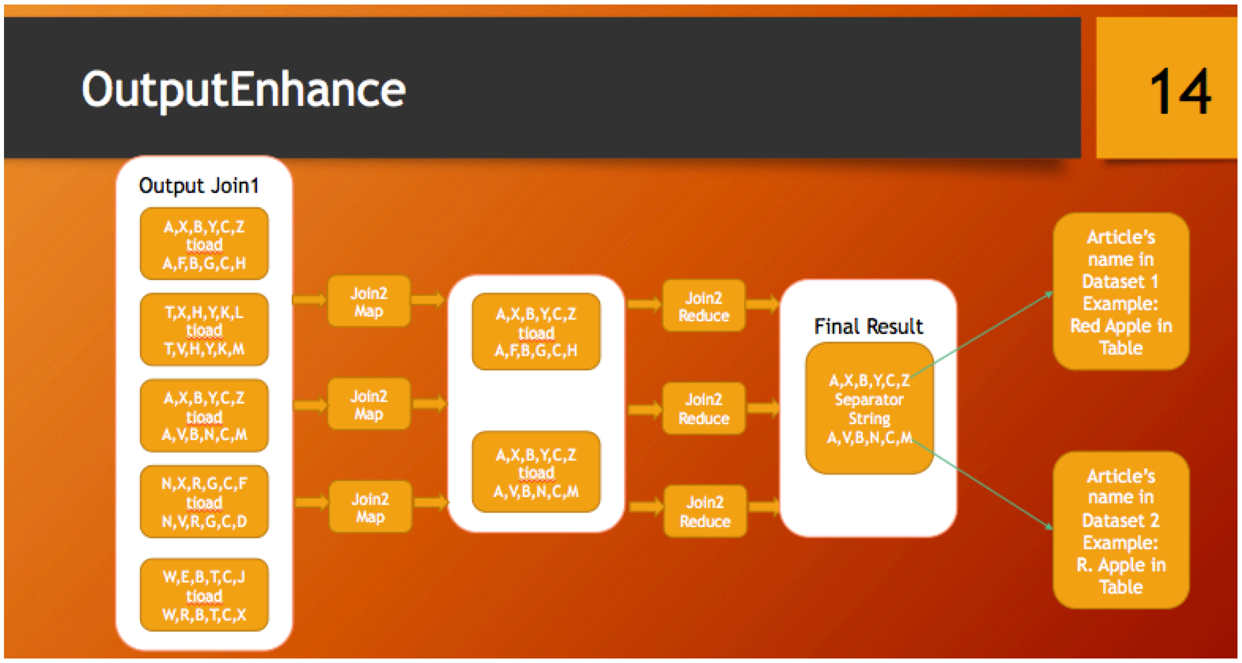

OutputEnhance
14
Output Join1
A,X,B,Y,C,Z tload A,F,B,G,C,H
T,X,H,Y,K,L tload T,V,H,Y,K,M
A,X,B,Y,C,Z tload A,V,B,N,C,M
N,X,R,G,C,F tload N,V,R,G,C,D
W,E,B,T,C,J tload W,R,B,T,C,X
Join2 Map
Join2 Map
Join2 Map
A,X,B,Y,C,Z tload A,F,B,G,C,H
A,X,B,Y,C,Z tload A,V,B,N,C,M
Join2 Reduce
Join2 Reduce
Join2 Reduce
Final Result
A,X,B,Y,C,Z Separator String A,V,B,N,C,M
Article's name in Dataset 1 Example: Red Apple in Table
Article's name in Dataset 2 Example: R. Apple in Table

Figure 10: Data flow of OutputEnhance

The fourth MapReduce job deals with a problem that one record from one dataset can be matched with multiple records from the other dataset. We take only one mapping and leave out all the rest.

4.5 OUR CUSTOM INPUT FORMAT - ELODUINPUTFORMAT

Using the result of 4.4 we make a custom input format to read the two datasets. This will make processing on those datasets more accurate, since whenever the program reads in R.a it will know that it's actually S.a although the two might be different. With the default TextInputFormat the program will think that they are two different records, and processing can give inaccurate results.

4.6 AGGREGATION

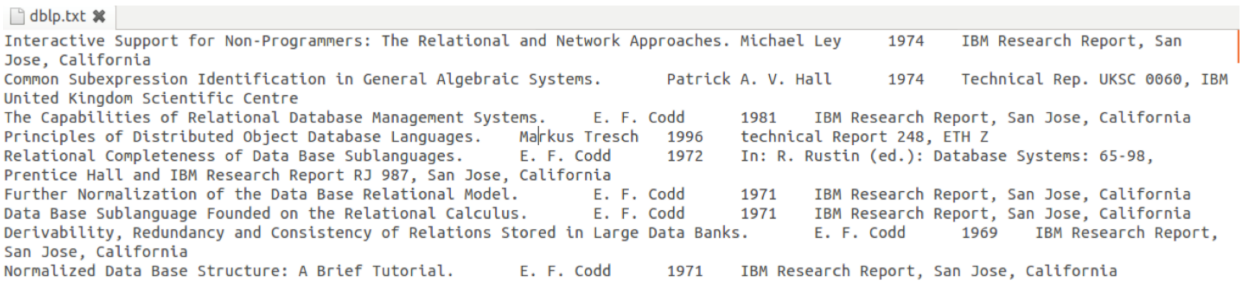
First we try joining the datasets using the default input format and our custom input format. Then we try grouping the records by target field and count the number of records grouped, also with the two input formats. Our custom input format works fine as the program doesn't miss records even when the same records appear differently in the two datasets.

5. EVALUATION

5.1 DATASET

Dataset is the first stage of this project and it is also one of the most troublesome part of our program. It is a tool for testing our project and guarantee that our solution can run and process fine with large data. This project will use two big datasets that was created from real datasets DBLP and CiteseerX.

5.1.1 DATASET 1

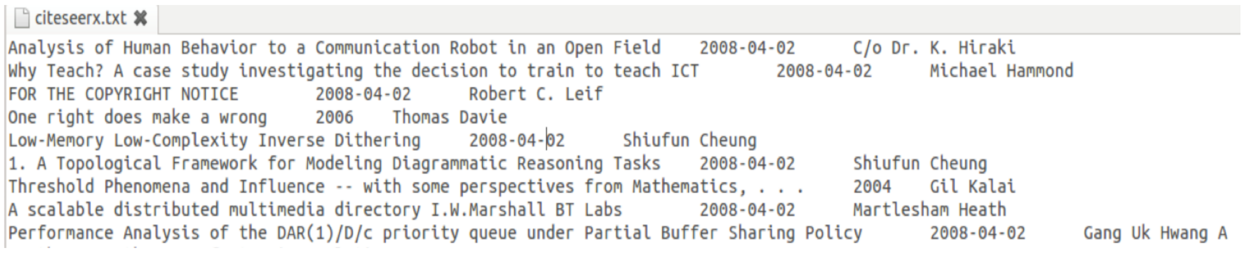


```
Interactive Support for Non-Programmers: The Relational and Network Approaches. Michael Ley     1974    IBM Research Report, San
Jose, California
Common Subexpression Identification in General Algebraic Systems.       Patrick A. V. Hall      1974    Technical Rep. UKSC 0060, IBM
United Kingdom Scientific Centre
The Capabilities of Relational Database Management Systems.     E. F. Codd      1981    IBM Research Report, San Jose, California
Principles of Distributed Object Database Languages.    Markus Tresch   1996    technical Report 248, ETH Z
Relational Completeness of Data Base Sublanguages.      E. F. Codd      1972    In: R. Rustin (ed.): Database Systems: 65-98,
Prentice Hall and IBM Research Report RJ 987, San Jose, California
Further Normalization of the Data Base Relational Model.        E. F. Codd      1971    IBM Research Report, San Jose, California
Data Base Sublanguage Founded on the Relational Calculus.       E. F. Codd      1971    IBM Research Report, San Jose, California
Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks.       E. F. Codd      1969    IBM Research Report,
San Jose, California
Normalized Data Base Structure: A Brief Tutorial.       E. F. Codd      1971    IBM Research Report, San Jose, California
```

Figure 11: Picture of Dataset 1

Dataset 1 is created base on a real big dataset called DBLP. The original dataset is very big with the size of 1.64 GB and in XML format. It is too big and have many unnecessary parts. Therefore, we write a Java program that converts XML format to normal text format, removes junk parts and makes dataset fit better with our program. After processing, the final dataset 1 is only 149 MB with more than 1 million records. The format of dataset 1 is showed on picture above.

5.1.2 DATASET 2



citeseerx.txt

```
Analysis of Human Behavior to a Communication Robot in an Open Field    2008-04-02      C/o Dr. K. Hiraki
Why Teach? A case study investigating the decision to train to teach ICT        2008-04-02      Michael Hammond
FOR THE COPYRIGHT NOTICE        2008-04-02      Robert C. Leif
One right does make a wrong     2006    Thomas Davie
Low-Memory Low-Complexity Inverse Dithering     2008-04-02      Shiufun Cheung
1. A Topological Framework for Modeling Diagrammatic Reasoning Tasks    2008-04-02      Shiufun Cheung
Threshold Phenomena and Influence -- with some perspectives from Mathematics, . . .     2004    Gil Kalai
A scalable distributed multimedia directory I.W.Marshall BT Labs        2008-04-02      Martlesham Heath
Performance Analysis of the DAR(1)/D/c priority queue under Partial Buffer Sharing Policy       2008-04-02      Gang Uk Hwang A
```

Figure 12: Picture of Dataset 2

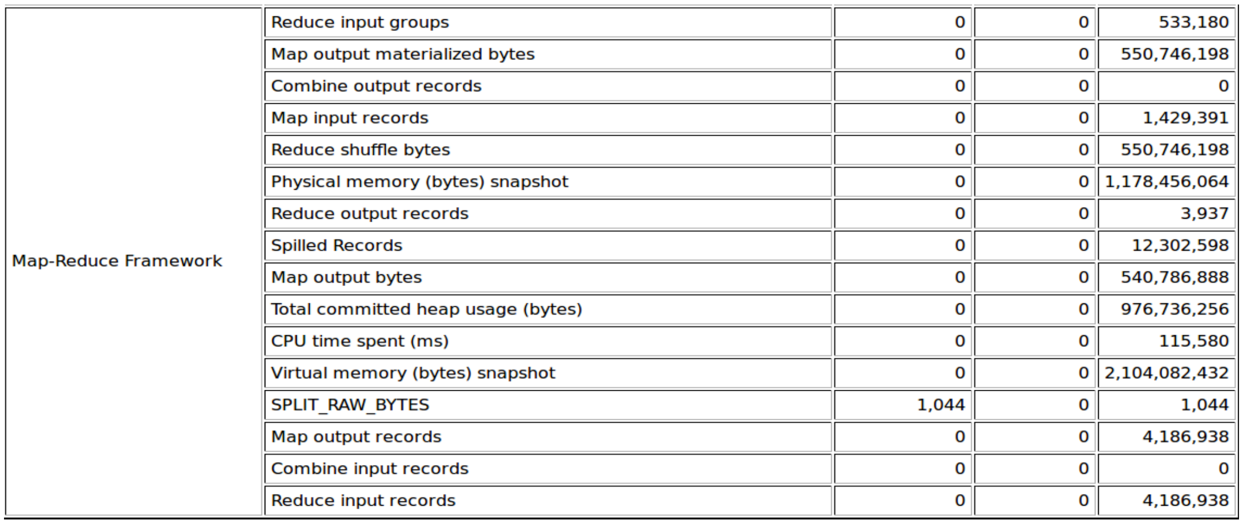
Dataset 2 is created base on a real big dataset called CiteseerX. The original dataset is large with the size of 7.6 GB and also in XML format. It is bigger than original DBLP dataset seven times and it has many unclear characters. Therefore, we only take one of the clearest parts on original CiteseerX dataset to process. We wrote a Java program that converts XML format to normal text format, removes junk parts and makes dataset fit better with our program. After processing, the size of final dataset 2 is 141 MB.

5.2 PERFORMANCE

After working and testing with two sample datasets (Dataset 1: 149 MB, Dataset 2: 141 MB) that we created base on DBLP and CiteSeerX above, we have achieved some very good results and this has shown the feasibility of our project.

5.2.1 SIMILARITY-JOIN TEST

Result of similarity-join using supporting fields is shown in the image below:

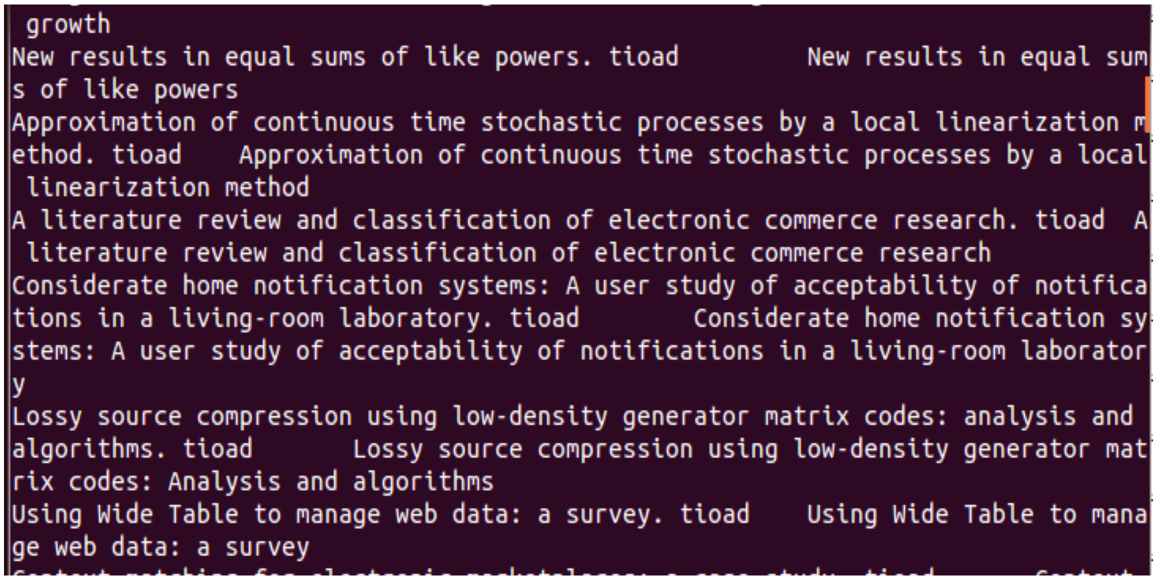


Map-Reduce Framework	Reduce input groups	0	0	533,180
	Map output materialized bytes	0	0	550,746,198
	Combine output records	0	0	0
	Map input records	0	0	1,429,391
	Reduce shuffle bytes	0	0	550,746,198
	Physical memory (bytes) snapshot	0	0	1,178,456,064
	Reduce output records	0	0	3,937
	Spilled Records	0	0	12,302,598
	Map output bytes	0	0	540,786,888
	Total committed heap usage (bytes)	0	0	976,736,256
	CPU time spent (ms)	0	0	115,580
	Virtual memory (bytes) snapshot	0	0	2,104,082,432
	SPLIT_RAW_BYTES	1,044	0	1,044
	Map output records	0	0	4,186,938
	Combine input records	0	0	0
	Reduce input records	0	0	4,186,938

Figure 13: Similarity-Join result statistics

There are more than 1.4 million records for input. The reduce output records are the results of the Similarity-Join job. The table shows 3937 records and the job took about 115,580 milliseconds (almost 2 minutes) to run through and process two big datasets.

The picture below will show one part of the result:

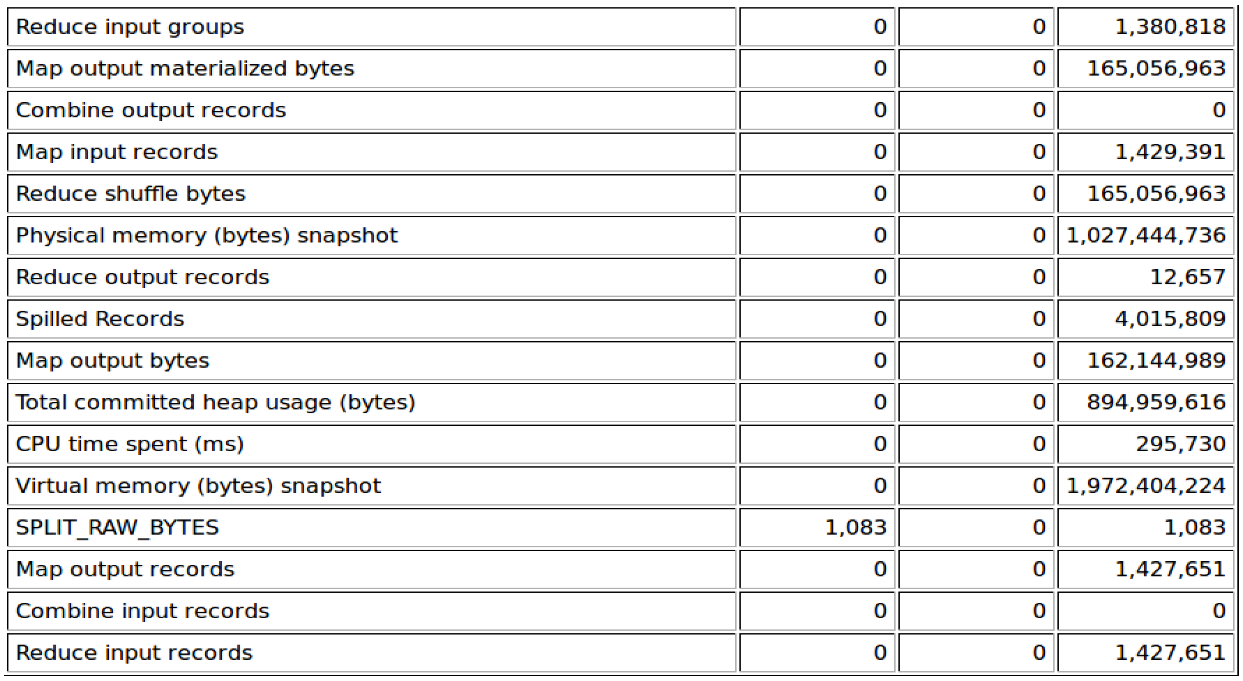


```
 growth
New results in equal sums of like powers. tioad        New results in equal sum
s of like powers
Approximation of continuous time stochastic processes by a local linearization m
ethod. tioad    Approximation of continuous time stochastic processes by a local
 linearization method
A literature review and classification of electronic commerce research. tioad  A
 literature review and classification of electronic commerce research
Considerate home notification systems: A user study of acceptability of notifica
tions in a living-room laboratory. tioad        Considerate home notification sy
stems: A user study of acceptability of notifications in a living-room laborator
y
Lossy source compression using low-density generator matrix codes: analysis and
algorithms. tioad       Lossy source compression using low-density generator mat
rix codes: Analysis and algorithms
Using Wide Table to manage web data: a survey. tioad    Using Wide Table to mana
ge web data: a survey
```

Figure 14: Sample of similarity join result

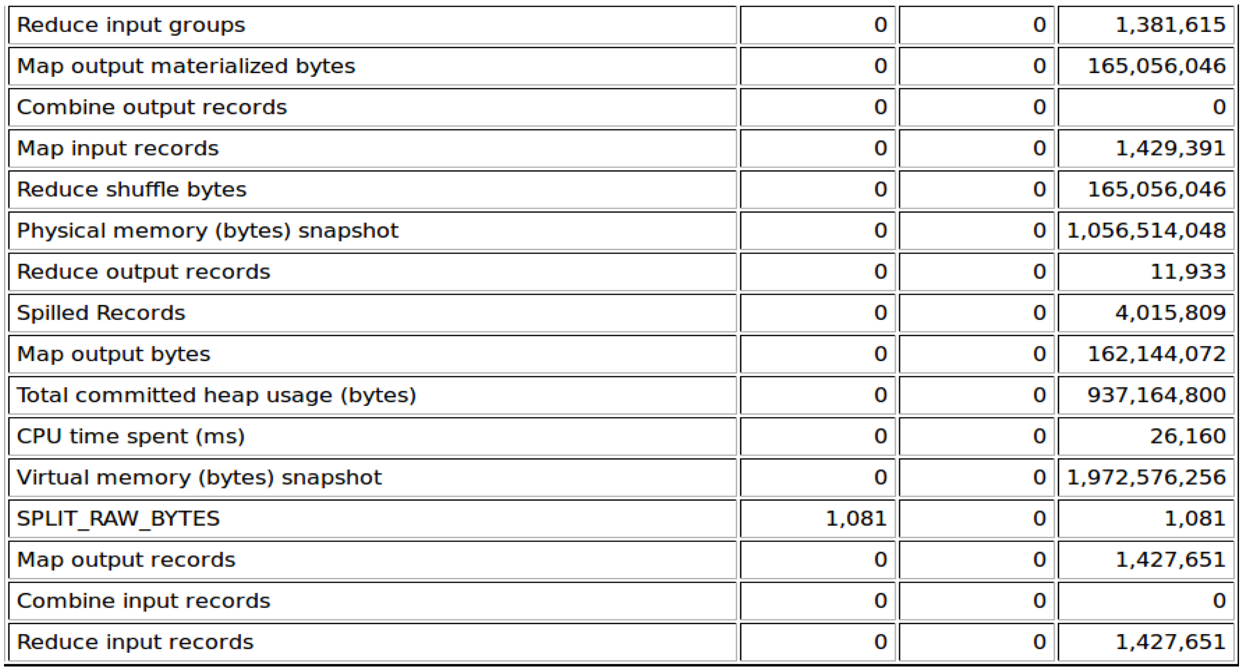
The result is fairly accurate. After doing the similarity join we will process out result through a filter to deal with duplicate records. The enhanced result after processing through filter is about 950 records. After testing the join part, we started to test the ELODUInputFormat against the TextInputFormat.

5.2.2 ELODUINPUTFORMAT VS. TEXTINPUTFORMAT IN JOIN



Reduce input groups	0	0	1,380,818
Map output materialized bytes	0	0	165,056,963
Combine output records	0	0	0
Map input records	0	0	1,429,391
Reduce shuffle bytes	0	0	165,056,963
Physical memory (bytes) snapshot	0	0	1,027,444,736
Reduce output records	0	0	12,657
Spilled Records	0	0	4,015,809
Map output bytes	0	0	162,144,989
Total committed heap usage (bytes)	0	0	894,959,616
CPU time spent (ms)	0	0	295,730
Virtual memory (bytes) snapshot	0	0	1,972,404,224
SPLIT_RAW_BYTES	1,083	0	1,083
Map output records	0	0	1,427,651
Combine input records	0	0	0
Reduce input records	0	0	1,427,651

Figure 15: ELODUInput format - Join result statistics

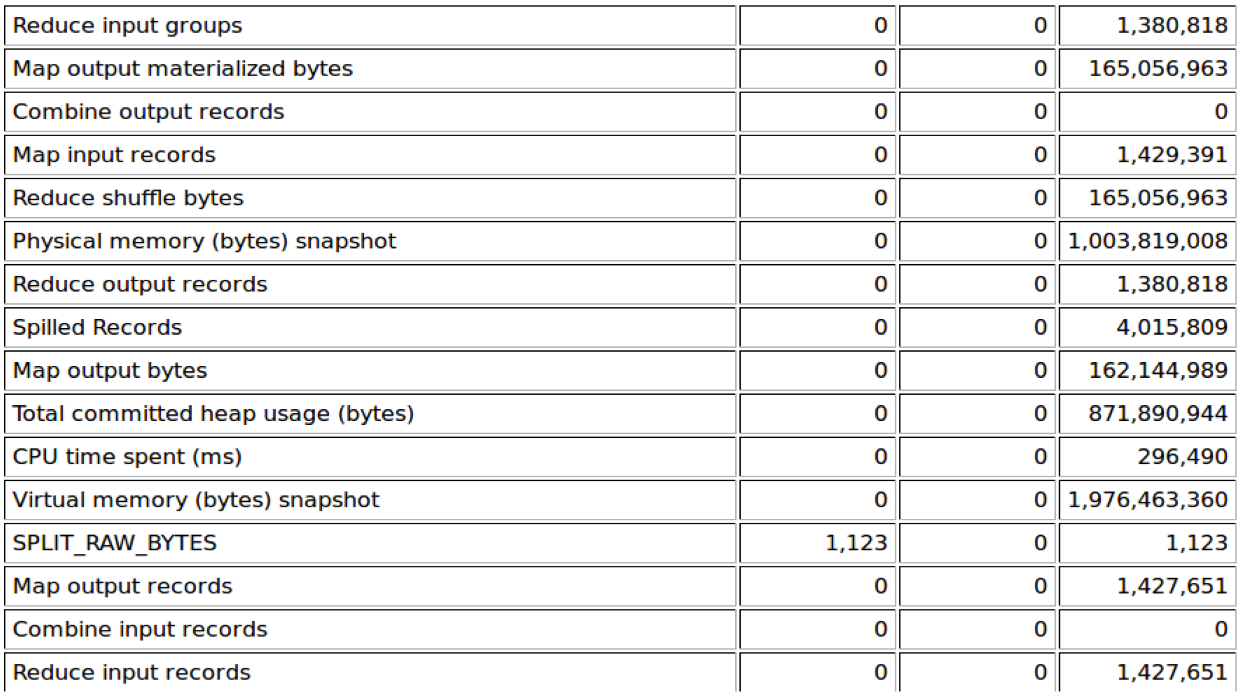


Reduce input groups	0	0	1,381,615
Map output materialized bytes	0	0	165,056,046
Combine output records	0	0	0
Map input records	0	0	1,429,391
Reduce shuffle bytes	0	0	165,056,046
Physical memory (bytes) snapshot	0	0	1,056,514,048
Reduce output records	0	0	11,933
Spilled Records	0	0	4,015,809
Map output bytes	0	0	162,144,072
Total committed heap usage (bytes)	0	0	937,164,800
CPU time spent (ms)	0	0	26,160
Virtual memory (bytes) snapshot	0	0	1,972,576,256
SPLIT_RAW_BYTES	1,081	0	1,081
Map output records	0	0	1,427,651
Combine input records	0	0	0
Reduce input records	0	0	1,427,651

Figure 16: TextInputFormat - Join result statistics

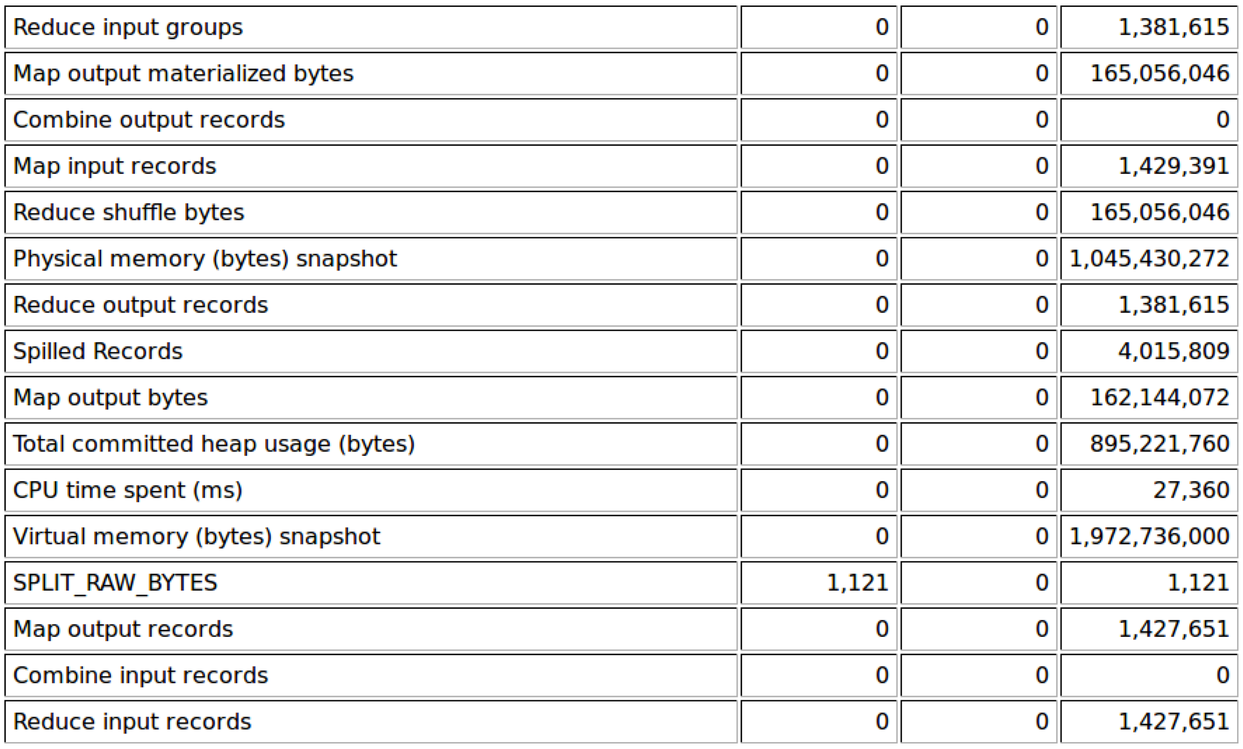
The result of ELODUInputFormat shows that the join takes about 295 seconds to process, higher than the join using TextInputFormat which takes only 26 seconds to finish. This because the ELODU input format needs to check the Similarity-Join result to find the matching formats for all records in the citeseerx data file. However the result of shows that the ELODUInputFormat finds more records matching than the default TextInputFormat since the program is capable of detecting matches even when the target field is not exactly the same.

5.2.3 ELODUINPUTFORMAT VS. TEXTINPUTFORMAT IN AGGREGATION



Reduce input groups	0	0	1,380,818
Map output materialized bytes	0	0	165,056,963
Combine output records	0	0	0
Map input records	0	0	1,429,391
Reduce shuffle bytes	0	0	165,056,963
Physical memory (bytes) snapshot	0	0	1,003,819,008
Reduce output records	0	0	1,380,818
Spilled Records	0	0	4,015,809
Map output bytes	0	0	162,144,989
Total committed heap usage (bytes)	0	0	871,890,944
CPU time spent (ms)	0	0	296,490
Virtual memory (bytes) snapshot	0	0	1,976,463,360
SPLIT_RAW_BYTES	1,123	0	1,123
Map output records	0	0	1,427,651
Combine input records	0	0	0
Reduce input records	0	0	1,427,651

Figure 17: ELODUInputformat - Aggregation result statistics



Reduce input groups	0	0	1,381,615
Map output materialized bytes	0	0	165,056,046
Combine output records	0	0	0
Map input records	0	0	1,429,391
Reduce shuffle bytes	0	0	165,056,046
Physical memory (bytes) snapshot	0	0	1,045,430,272
Reduce output records	0	0	1,381,615
Spilled Records	0	0	4,015,809
Map output bytes	0	0	162,144,072
Total committed heap usage (bytes)	0	0	895,221,760
CPU time spent (ms)	0	0	27,360
Virtual memory (bytes) snapshot	0	0	1,972,736,000
SPLIT_RAW_BYTES	1,121	0	1,121
Map output records	0	0	1,427,651
Combine input records	0	0	0
Reduce input records	0	0	1,427,651

Figure 18: TextInput format - Aggregation result statistics

With the Aggregation use case, the CPU time to finish the job is quite the same with the Join. However the program that uses ELODUInputFormat output less records than the one using TextInputFormat. This is because ELODUInputFormat detected more matchings, so more records was grouped together.

6. CONCLUSION

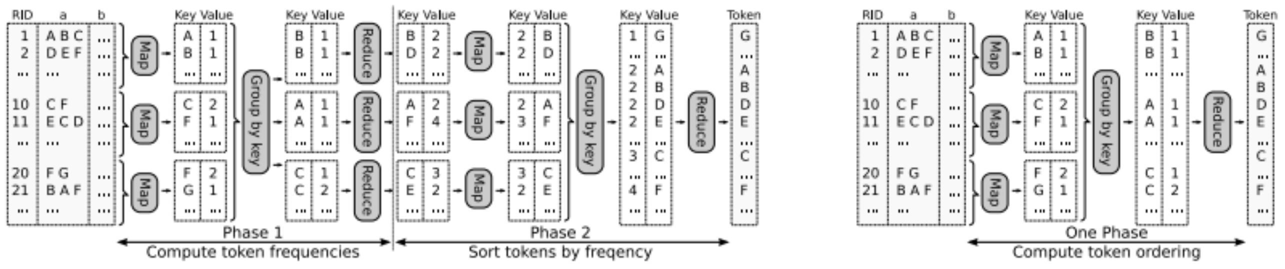
This project provides a new way to handle the situation of processing data with redundancy and bad formats which happens so oftenly. Our method can make processing these kinds of data more accurate. Overall, we accomplished all objectives for this project. However, for applying to reality the project needs much improvement especially on the execution speed. We believe that in the very near future, out project will be a powerful method for Entity Resolution in Big Data.

7. RELATED WORKS

7.1 EFFICIENT PARALLEL SET-SIMILARITY JOINS USING MAPREDUCE – UNIVERSITY OF CALIFORNIA, IRVINE

Identifying similar pair of records is applied in many applications and our project is one of them. "Efficient Parallel Set-Similarity Joins Using MapReduce" paper is written by Rares Vernica, Michael J. Carey and Chen Li – Department of Computer Science in University of California, Irvine. This paper provides some effective algorithms to dealing with end-to-end set-similarity-join problems through 3-stage approach, and our Similarity-Join job is based on the technique presented in this paper.

Stage 1: Dividing the data into tokens and order them base on their number of occurences in the datasets. The next stage will use the result of token ordering to compute the similarity. Our Similarity-Join job used the Basic Token Ordering method, which consists of two main parts. The first part computes the tokens' frequencies and the second part sorts tokens by their frequencies.



(a) Basic Token Ordering (BTO)

(b) One-Phase Token Ordering (OPTO)

Figure 16: Data flow of Stage One [2]

Stage 2: Taking two datasets as input, assign record ID for each record from two datasets, identify the join attribute value of each record. Using the result from Token Ordering part to identify, pair the RIDs and the join attributes from two datasets. The pairs that are in common will go to the same reducer. The reducer will compute the similarity of the values (join attribute) and return RIDs of records that are similar. Our Project used the idea but instead of producing records IDs, we produced the target field pairs that passed the matching test, because our purpose is to use this type of result to make a custom input format.

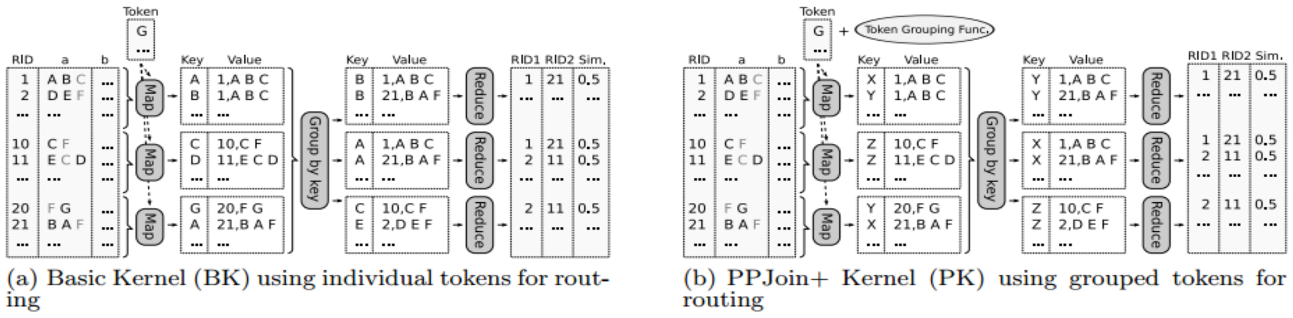

Token
Token Grouping Func.
RID
Key
Value
Map
Group by key
Reduce
RID1 RID2 Sim.
(a) Basic Kernel (BK) using individual tokens for routing
(b) PPJoin+ Kernel (PK) using grouped tokens for routing

Figure 17: Data flow of Stage Two [2]

Stage 3: Using the results from the second stage (pairs of RIDs) to find and build the actual pairs of records that are similar from the original datasets.

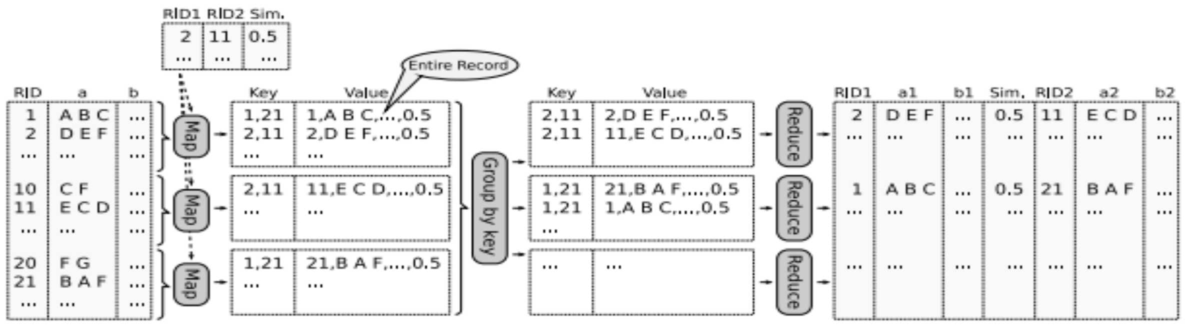


Figure 18: Data flow of Stage Three using One-Phase Record Join [2]

APPENDIX

APPENDIX A: COUNTING FREQUENCY JAVA CODE

Map function:

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {

    String whole = value.toString();
    String[] line = whole.split("\t",3);
          StringTokenizer itr = new StringTokenizer(line[0]);
          while (itr.hasMoreTokens()) {
            String ntk = itr.nextToken();
            word.set(ntk);
            context.write(word, one);
          }
    }
```

Reduce function:

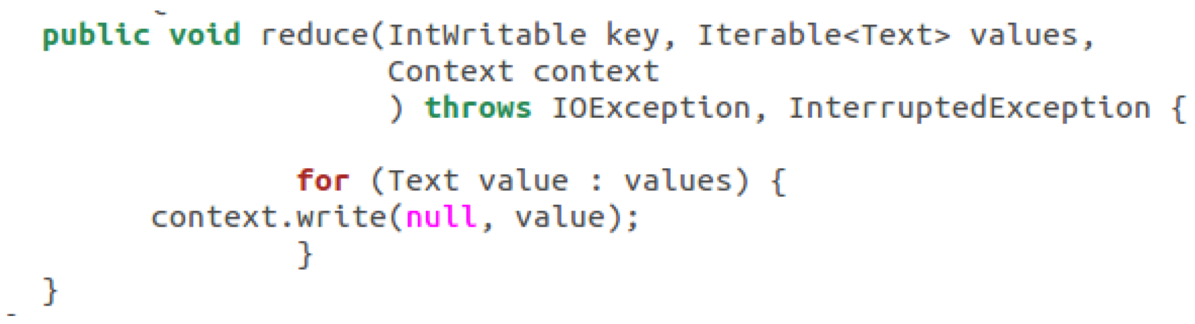
```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

APPENDIX B: ORDERING TOKENS JAVA CODE

Map function:

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
    String whole = value.toString();
    String[] line = whole.split("\t", 2);

    freq.set(Integer.parseInt(line[1]));
    word.set(line[0]);
    context.write(freq, word);
    }
```

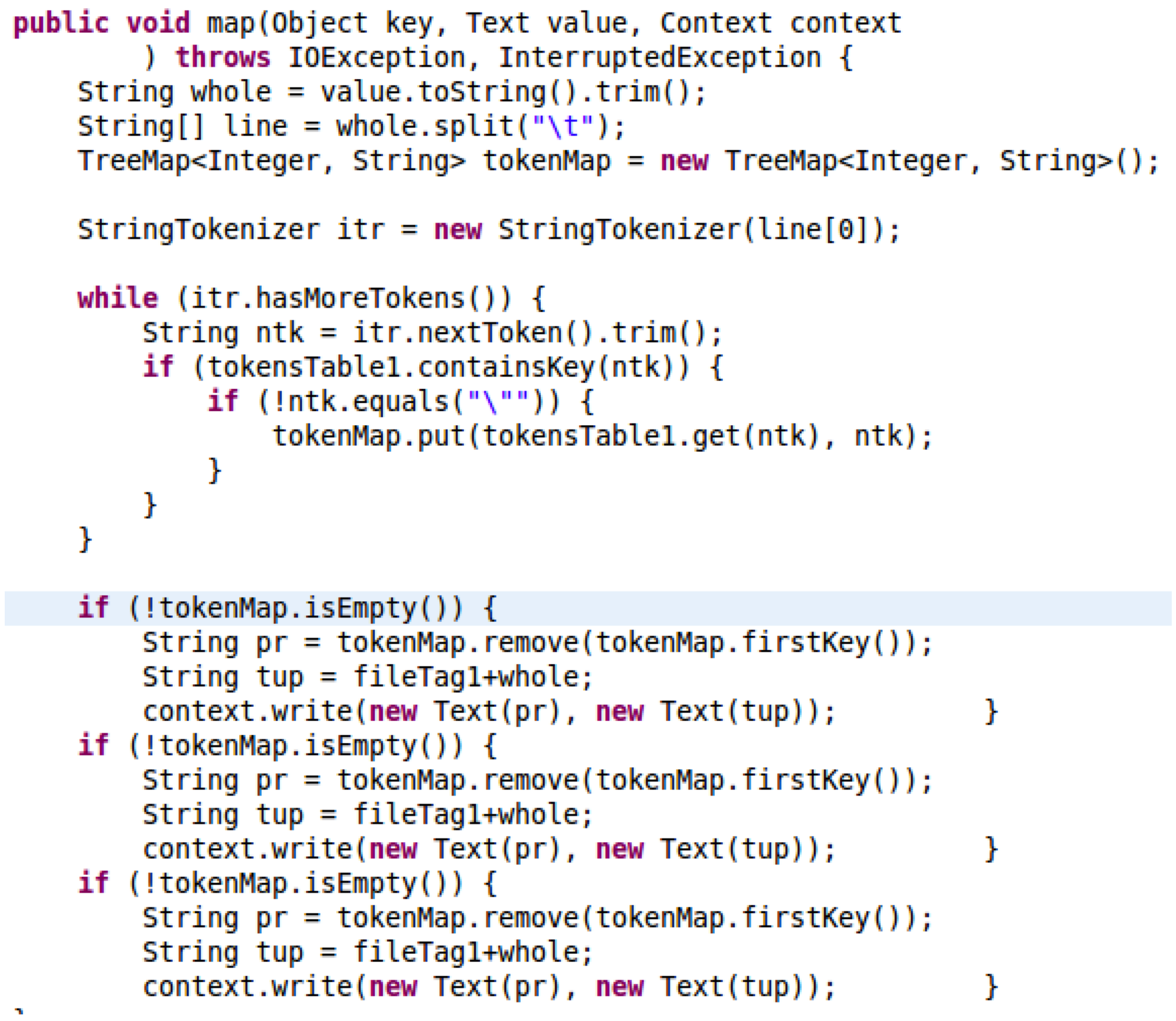
Reduce function:



```
    public void reduce(IntWritable key, Iterable<Text> values,
                       Context context
                       ) throws IOException, InterruptedException {

                  for (Text value : values) {
        context.write(null, value);
                  }
    }
```

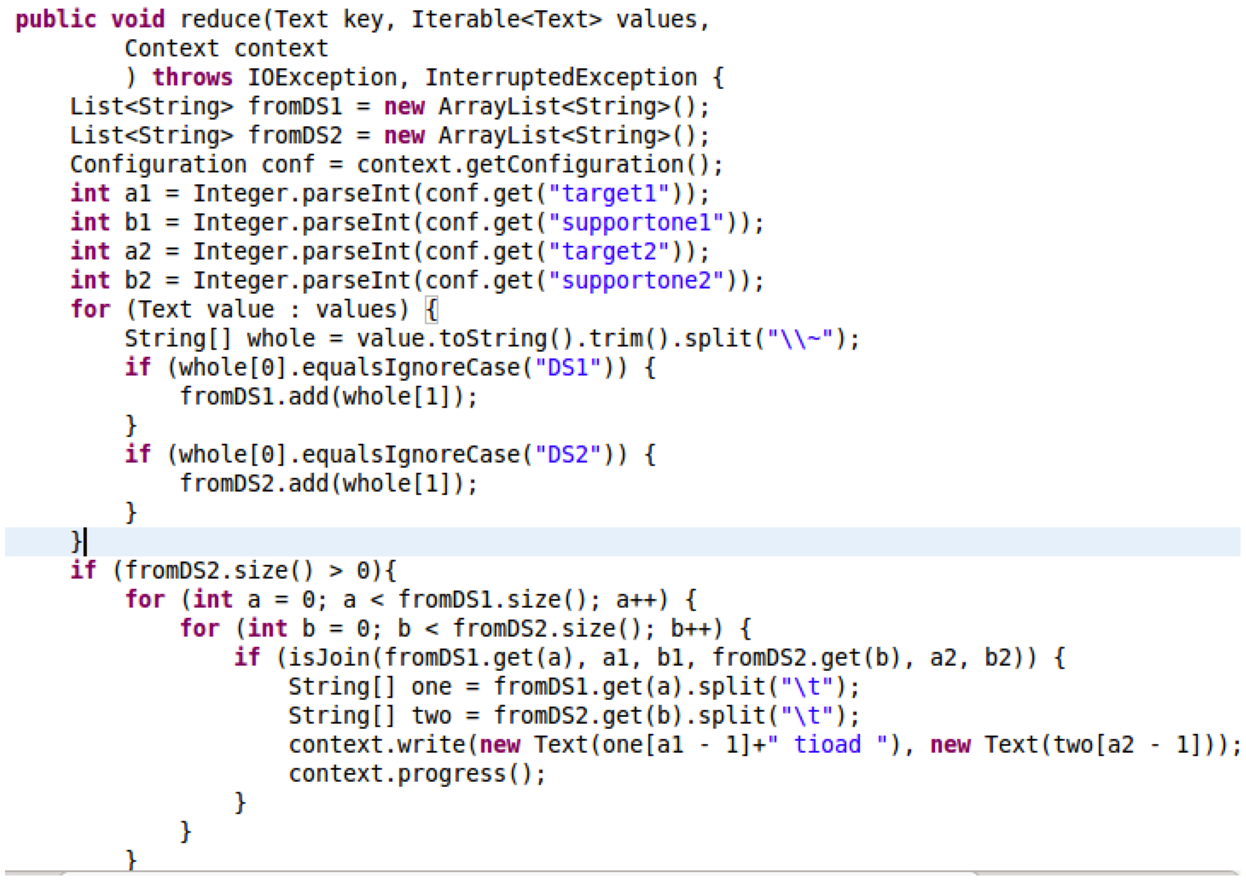
APPENDIX C: SIMILARITY JOIN JAVA CODE

Map function:



```
public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
    String whole = value.toString().trim();
    String[] line = whole.split("\t");
    TreeMap<Integer, String> tokenMap = new TreeMap<Integer, String>();

    StringTokenizer itr = new StringTokenizer(line[0]);

    while (itr.hasMoreTokens()) {
        String ntk = itr.nextToken().trim();
        if (tokensTable1.containsKey(ntk)) {
            if (!ntk.equals("\"")) {
                tokenMap.put(tokensTable1.get(ntk), ntk);
            }
        }
    }

    if (!tokenMap.isEmpty()) {
        String pr = tokenMap.remove(tokenMap.firstKey());
        String tup = fileTag1+whole;
        context.write(new Text(pr), new Text(tup));         }
    if (!tokenMap.isEmpty()) {
        String pr = tokenMap.remove(tokenMap.firstKey());
        String tup = fileTag1+whole;
        context.write(new Text(pr), new Text(tup));         }
    if (!tokenMap.isEmpty()) {
        String pr = tokenMap.remove(tokenMap.firstKey());
        String tup = fileTag1+whole;
        context.write(new Text(pr), new Text(tup));         }
```

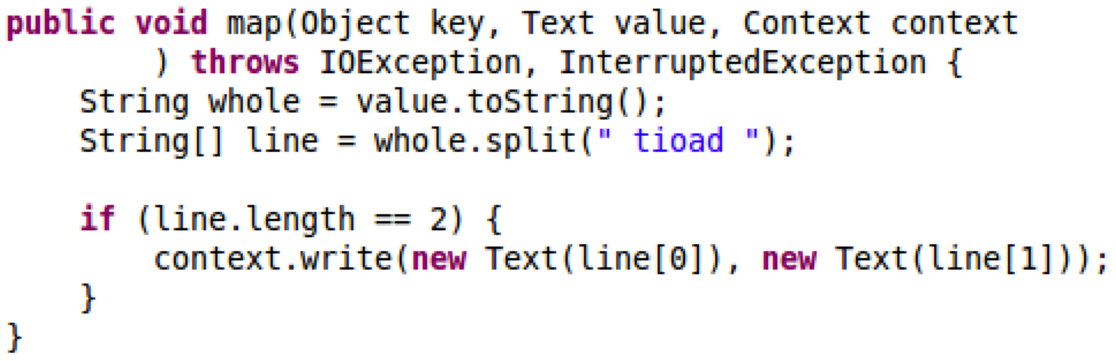
Reduce function:



```
public void reduce(Text key, Iterable<Text> values,
        Context context
        ) throws IOException, InterruptedException {
    List<String> fromDS1 = new ArrayList<String>();
    List<String> fromDS2 = new ArrayList<String>();
    Configuration conf = context.getConfiguration();
    int a1 = Integer.parseInt(conf.get("target1"));
    int b1 = Integer.parseInt(conf.get("supportone1"));
    int a2 = Integer.parseInt(conf.get("target2"));
    int b2 = Integer.parseInt(conf.get("supportone2"));
    for (Text value : values) {
        String[] whole = value.toString().trim().split("\\~");
        if (whole[0].equalsIgnoreCase("DS1")) {
            fromDS1.add(whole[1]);
        }
        if (whole[0].equalsIgnoreCase("DS2")) {
            fromDS2.add(whole[1]);
        }
    }
    if (fromDS2.size() > 0){
        for (int a = 0; a < fromDS1.size(); a++) {
            for (int b = 0; b < fromDS2.size(); b++) {
                if (isJoin(fromDS1.get(a), a1, b1, fromDS2.get(b), a2, b2)) {
                    String[] one = fromDS1.get(a).split("\t");
                    String[] two = fromDS2.get(b).split("\t");
                    context.write(new Text(one[a1 - 1]+" tioad "), new Text(two[a2 - 1]));
                    context.progress();
                }
            }
        }
```

APPENDIX D: ENHANCE RESULT JAVA CODE

Map function:



```
public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
    String whole = value.toString();
    String[] line = whole.split(" tioad ");

    if (line.length == 2) {
        context.write(new Text(line[0]), new Text(line[1]));
    }
}
```

Reduce function:

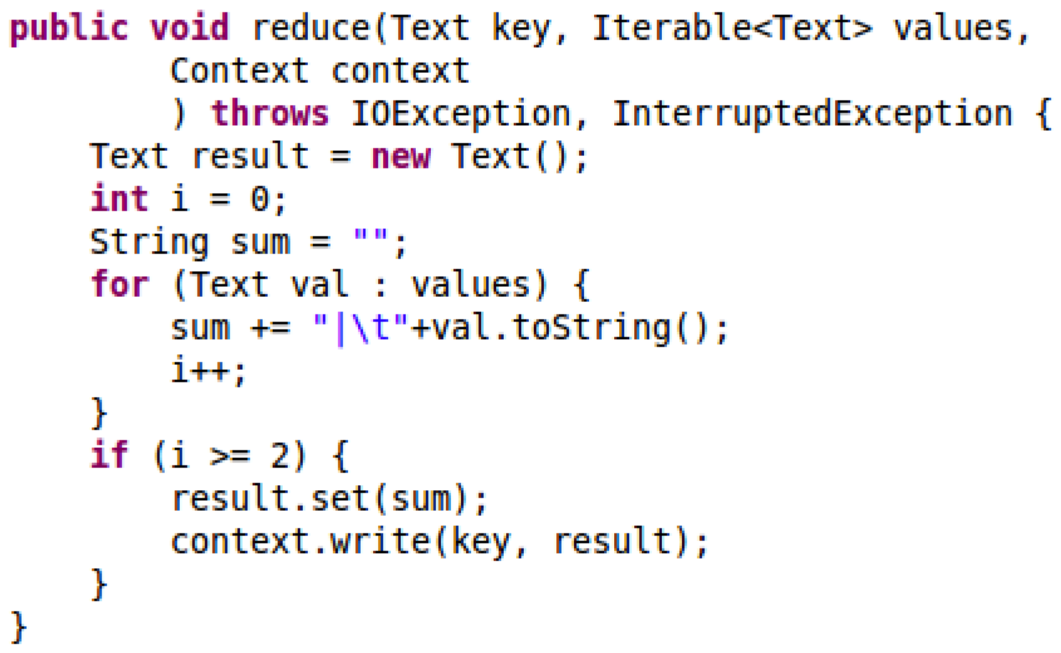
```
public void reduce(Text key, Iterable<Text> values,
        Context context
        ) throws IOException, InterruptedException {
    String whole = key.toString();

    Configuration conf = context.getConfiguration();
    String aString = "none";

    for (Text value : values) {
        String val = value.toString();
        if (!(whole.equalsIgnoreCase(val))) {
            aString = val;
        }
    }
    if (aString.equalsIgnoreCase("none")) {
    }
    else {
        context.write(new Text(whole+" separatorstring "), new Text(aString));
    }
}
```

APPENDIX E: JOINTEST JAVA CODE

Map function:

```
public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
    Text keytext = new Text();
    Text valuetext = new Text();
    String whole = value.toString();
    String[] line = whole.split("\t",2);
    if (line.length == 2) {
        keytext.set(line[0].trim());
        valuetext.set(line[1]);
        context.write(keytext, valuetext);
    }
}
```

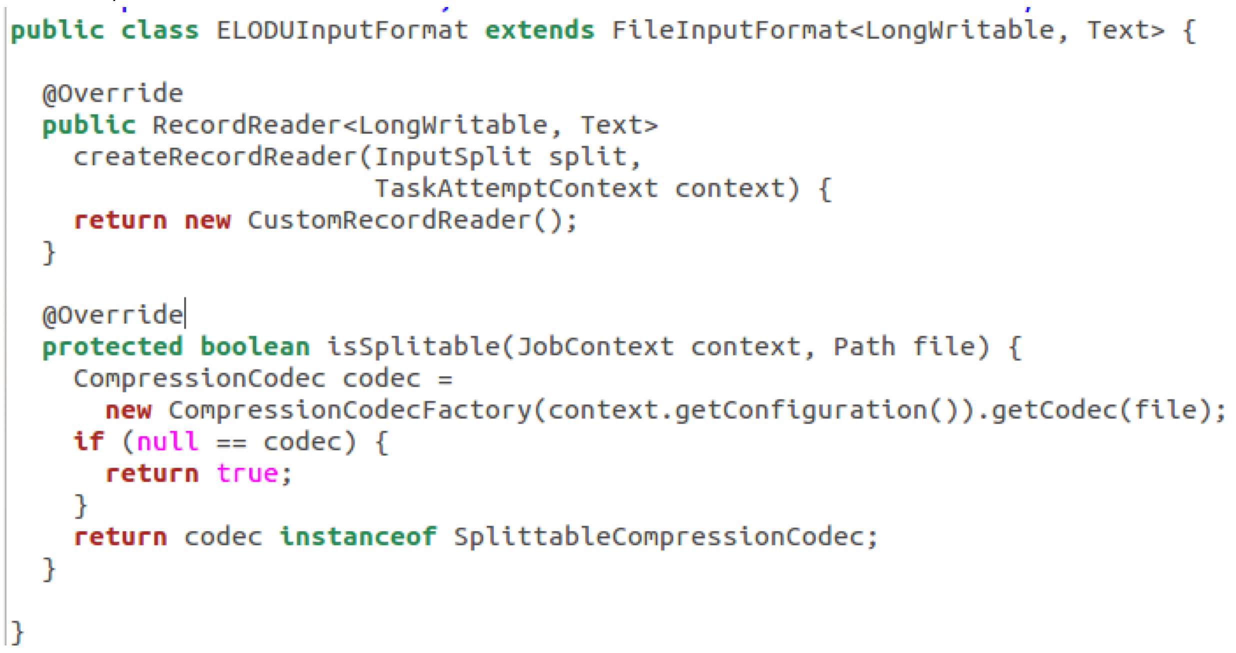
Reduce function:



```
public void reduce(Text key, Iterable<Text> values,
        Context context
        ) throws IOException, InterruptedException {
    Text result = new Text();
    int i = 0;
    String sum = "";
    for (Text val : values) {
        sum += "|\t"+val.toString();
        i++;
    }
    if (i >= 2) {
        result.set(sum);
        context.write(key, result);
    }
}
```

APPENDIX F: CUSTOM INPUT FORMAT JAVA CODE

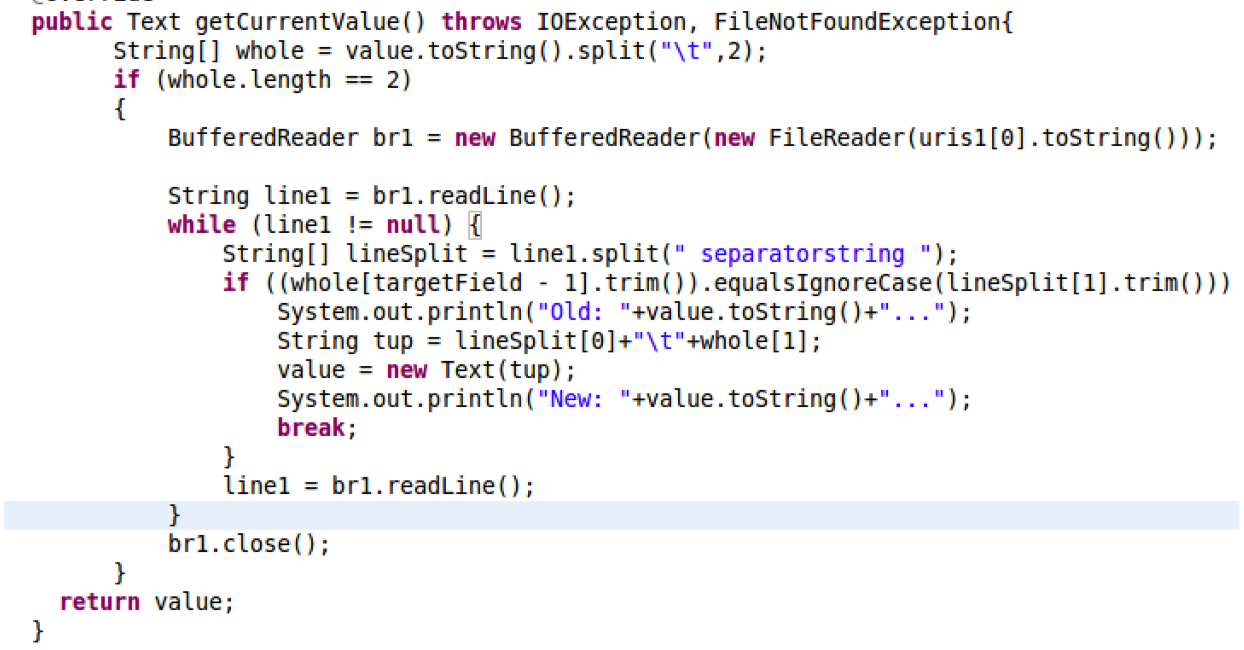
ELODUInputFormat class:



```
public class ELODUInputFormat extends FileInputFormat<LongWritable, Text> {

  @Override
  public RecordReader<LongWritable, Text>
    createRecordReader(InputSplit split,
                       TaskAttemptContext context) {
    return new CustomRecordReader();
  }

  @Override
  protected boolean isSplitable(JobContext context, Path file) {
    CompressionCodec codec =
      new CompressionCodecFactory(context.getConfiguration()).getCodec(file);
    if (null == codec) {
      return true;
    }
    return codec instanceof SplittableCompressionCodec;
  }

}
```

APPENDIX G: CUSTOM RECORD READER JAVA CODE

getCurrentValue() method:



```
public Text getCurrentValue() throws IOException, FileNotFoundException{
      String[] whole = value.toString().split("\t",2);
      if (whole.length == 2)
      {
          BufferedReader br1 = new BufferedReader(new FileReader(uris1[0].toString()));

          String line1 = br1.readLine();
          while (line1 != null) {
              String[] lineSplit = line1.split(" separatorstring ");
              if ((whole[targetField - 1].trim()).equalsIgnoreCase(lineSplit[1].trim())){
                  System.out.println("Old: "+value.toString()+"...");
                  String tup = lineSplit[0]+"\t"+whole[1];
                  value = new Text(tup);
                  System.out.println("New: "+value.toString()+"...");
                  break;
              }
              line1 = br1.readLine();
          }
          br1.close();
      }
  return value;
}
```

CITATIONS

[1] Apache™. (2014). Welcome to The Apache™ Hadoop®!. The Apache Software Foundation. https://hadoop.apache.org/index.pdf

[2] Vernica, R. Carey, M. Li, C. (2010). Efficient Parallel Set-Similarity Joins Using MapReduce. University of California, Irvine. Department of Computer Science. http://flamingo.ics.uci.edu/pub/sigmod10-vernica.pdf

[3] http://dblp.uni-trier.de/xml/dblp.xml.gz

[4] http://csxstatic.ist.psu.edu/about