Spring 2016

# Hybrid Similarity Function for Big Data Entity Matching with R-Swoosh

Vimal Chandra Gorijala
*San Jose State University*

# Hybrid Similarity Function for Big Data Entity Matching with R-Swoosh

A Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfilment
of the Requirements for the Degree
Master of Science

by
Vimal Chandra Gorijala
February 2016

The Designated Project Committee Approves the Project Titled

Hybrid Similarity Function for Big Data Entity Matching
with R-Swoosh

by
Vimal Chandra Gorijala

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

February 2016

| | |
|---|---|
| Dr. Thanh Tran | Department of Computer Science |
| Dr. Robert Chun | Department of Computer Science |
| Mr. Sharath Chandra Pilli | Sr. Software Engineer at Captora |

## ABSTRACT

### Hybrid Similarity Function for Big Data Entity Matching with R-Swoosh

**by Vimal Chandra Gorijala**

Entity Matching (EM) is the problem of determining if two entities in a data set refer to the same real-world object. For example, it decides if two given mentions in the data, such as “Helen Hunt” and “H. M. Hunt”, refer to the same real-world entity by using different similarity functions. This problem plays a key role in information integration, natural language understanding, information processing on the World-Wide Web, and on the emerging Semantic Web. This project deals with the similarity functions and thresholds utilized in them to determine the similarity of the entities. The work contains two major parts: implementation of a hybrid similarity function, which contains three different similarity functions to determine the similarity of entities, and an efficient method to determine the optimum threshold value for similarity functions to get accurate results.

## ACKNOWLEDGEMENTS

Firstly, I would like to take this opportunity to thank my project advisor, Dr. Thanh Tran, for his guidance throughout the project. It would not have been possible without his support throughout the project.

I would also like to thank my committee members Dr. Robert Chun and Mr. Sharath Chandra Pilli, for their valuable advices and comments during project report writing phase.

Furthermore, I would like to thank my brother and parents for always being there for me during my master's program. Last but not the least; I would like to thank all my friends for their tremendous support throughout the completion of this project.

**Table of Contents**

## LIST OF TABLES



## LIST OF FIGURES

## ACRONYMS

Table 1: Acronyms [4]

| Notations | Descriptions |
|---|---|
| $R$ | a relation |
| $a$ | an attribute in $R$ |
| $r$ | a record in $R$ |
| $\lambda$ | an attribute-matching rule (AR) |
| $\lambda^i$ | an implicit attribute-matching rule (iAR) |
| $\lambda^e$ | an explicit attribute-matching rule (eAR) |
| $\phi$ | a record-matching rule (RR) |
| $\psi$ | an explicit record-matching rule (eRR) |
| $\Phi$ | a set of record-matching rules (RRs) |
| $\Psi$ | a set of explicit record-matching rules (eRRs) |
| $f$ | a similarity function |
| $\mathcal{F}$ | a set of similarity functions |
| $\theta$ | a similarity-function threshold |
| $\Theta$ | a threshold range |
| $E$ | a set of examples |
| $M$ | a set of positive examples |
| $D$ | a set of negative examples |
| $\overline{M}_\Psi$ | a set of record pairs generated by $\Psi$ |

# CHAPTER 1

## Introduction

In Entity Matching, structured data is given as input and compared with the entities in the knowledge base and matching entities are identified in a ranked order. For instance, the mailing lists may have different entries referring to the same physical address, but the entries may be slightly different with different spellings or missing some information. Using various matching techniques, the matching entries are found. The knowledge base has both structured and unstructured datasets. Solved in a generic manner, entity matching will take O (n*n) comparisons. For instance, if there 10 million records available it would take 100 million comparisons to find the matching records. To calculate the matching entities in effective manner techniques like parallel implementation, dedoop, etc. are being used for large scale entity matching. The following are sub-tasks involved in large scale entity matching:

i) **Evaluation:** This task focuses on assessing the quality of the data in the knowledge base like attributes of the datasets, matching attributes, missing values for each attribute etc.

ii) **Pre-processing:** This task involves modifying the raw data using techniques like stemming, lemmatization, standardization, filling in missing values, stop words removal, attribute verification, conversion from upper case to lower case etc. This step increases the chances of finding the matching instances.

iii) **Candidate Calculation:** In this task, the matching candidates to the input given are calculated. For example, if we are searching for a person's bank account information it is inefficient to search all the data as there are many dissimilar candidates. To overcome this, similar candidates to the input are calculated using approaches like blocking.

Catch, these candidates contain roughly more than 80% similar candidates to the input records given.

iv) **Classification:** In here, the matching candidates from the above step are fed to a similarity function and classified as match and non-match.

This project offers a solution to the last step of the large-scale entity matching 'Classification'. The main aim is to classify the entities as match and non-match with good accuracy rate on a large scale.

This project involves implementation of two strategies: different similarity functions like Jaro similarity, edit-distance similarity, Jaccard similarity, etc. to classify the entities and a hybrid similarity function, which contains three similarity functions to classify the entities as match and non-match.

It also involves implementation of an efficient algorithm 'R-Swoosh' to reduce the number of comparisons while calculating the similarity score between entities. This algorithm reduces the unnecessary comparisons and decreases the time taken for execution. I have also implemented an efficient strategy to define the optimum threshold value for the similarity functions. By using this optimum threshold value, we can obtain better accuracy of results.

Real world practical datasets are used to evaluate this project. Apache Spark is used to implement the whole project in a distributed way and it can handle large-scale datasets.

The challenges regarding the parallel implementation of large-scale entity matching are addressed in the sections below.

Other concerns regarding previous steps to Classification in large scale entity matching are not in the scope of this project and therefore will not be discussed below.

# CHAPTER 2

## Related Works

This chapter consists of introduction to similarity functions, hybrid similarity function, Rule based matching and Swoosh algorithm.

### 2.1 Similarity Function

Similarity function is defined as a distance metric between different data points. This function calculates the distance score between the data points and compares it with a pre-defined threshold value. If the calculated distance is greater than the threshold value, the data points are a match otherwise they are considered as non-match. There are different types of similarity functions available. Based on the type of data points we can choose the similarity function. Some of them are:

**Jaccard Distance:** The Jaccard similarity coefficient is a statistic used for comparing the similarity and diversity of sample strings. [6] The Jaccard coefficient measures similarity between finite sample sets. [6] It is defined as the size of the intersection divided by the size of the union of the sample sets: J (A, B) = (|A ∩ B|/|A U B| (If A and B are both empty, we define J (A, B) = 1.) 0≤ J (A, B) ≤ 1. [6]

**Jaro Similarity Function:** The Jaro similarity defines ‘matching characters’ as characters in strings s1 and s2 that are (1) the same, and (2) whose indices are no farther than.[7] If m is the number of matching characters between strings x and y and t is the number of transpositions, the Jaro distance is defined as 1/3((m/x+m/y + (m-t/m)) when m is greater than 0, and 0 otherwise. [7]

**Edit Distance:** Given two strings ‘a’ and ‘b’, the edit distance d (a, b) is the minimum-weight series of edit operations that transforms ‘a’ into ‘b’. [5][8] Each of the operation has unit cost. Some of the simplest sets of edit operations are

Insertion of a single symbol: If a = uv, then inserting the symbol x produces uxv. [5][8]This can also be denoted $\varepsilon \rightarrow x$, using $\varepsilon$ to denote the empty string. [5][8]

Deletion: Deletion of a single symbol changes uxv to uv ($x \rightarrow \varepsilon$ ). [5][8]

Substitution: Substitution of a single symbol x for a symbol $y \neq x$ changes uxv to uyv ($x \rightarrow y$). [5][8]

### 2.2 Hybrid Similarity Function

The hybrid similarity function is combination of different similarity functions. This function calculates the distance between data points using different similarity functions in it. Then we take all the distances calculated and apply different techniques on them to make it into a single distance measure. This distance measure is compared with the pre-defined threshold value. If the calculated distance is greater than the threshold value, the data points are a match otherwise they are considered as non-match. The techniques used to combine different distances calculated are taking the average of them, assigning different weights to them basing on the importance, etc.

### 2.3 Rule Based Matching:

In rule based matching usually rules are defined basing on the attributes of the database we are dealing with. Initially a set of record matching rules are defined and basing on them, the similar records are found. For instance, a record matching rule can be “if the records have same telephone number and similar name, they are same.”

These record matching rules are defined mainly based on the database schema we are dealing with. If Schema changes, the rules which are previously defined does not apply. However, from these rules defined initially we can implement an efficient method to find the optimum threshold value for different similarity functions we utilize. The optimum threshold value is determined based on the observation different similarity functions and thresholds have redundancy. Based on it we can discard the inappropriate similarity functions.

### 2.4 Apache Spark Framework

For large scale Entity Matching Parallel programming is the best approach. The best example for the parallel programming is MapReduce model. Generally, MapReduce framework, Apache Spark, etc. are famous for parallel implementation of datasets. Spark can run on different file systems including HDFS. Now-a-days Apache Spark is considered more over MapReduce framework for parallel programming. Spark is 100x faster than Hadoop and it handles most of operations ‘in memory’, copying datasets from distributed physical storage into far faster logical RAM memory. [1] That is why I have chosen Apache Spark to implement the hybrid matcher function with R-Swoosh.

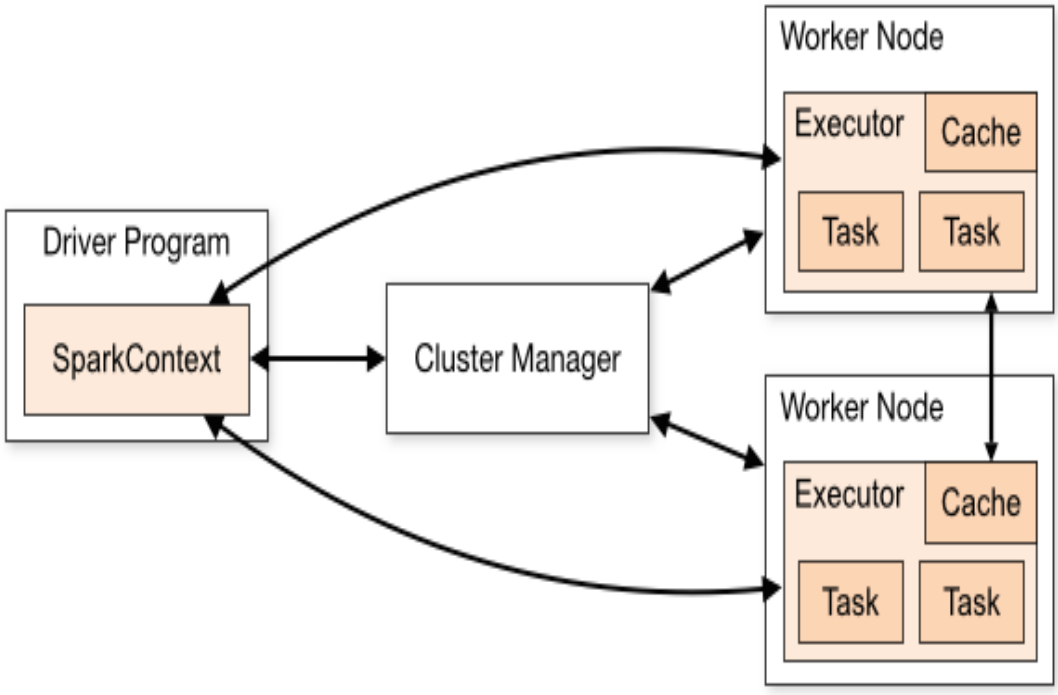


Figure 1: Apache Spark Architecture [1]

1. Each application gets its own executor processes. [1] They exist for whole duration of the application and runs the tasks in multiple threads. [1]
2. It supports the local mode by the whole setup in YARN cluster. [1] It can also be run in standalone mode. [1]
3. The driver should run closer to worker nodes as it schedules the tasks overall cluster. [1] Mainly jobs are submitted to the spark framework using the spark submit script. [1] The driver should be network accessible from the worker nodes. [1]

# CHAPTER 3

# DEFINITION, EXISTING SOLUTIONS AND PROPOSED SOLUTION

## 3.1 Definition:

### 3.1.1 Problem Formulation:

Given a large dataset, the steps before Classification in the large scale entity matching create candidate pairs (Entity pairs) with the help of efficient techniques like blocking. These candidate pairs are considered as worthy candidates for matching task. How to classify these pairs as match and non-match in a scalable and distributed manner in a minimal amount of time with good accuracy.

### 3.1.2 Terminology:

The following are the terms used in the report frequently.

Entity: It represents a concept or record, which has a meaning to itself completely. In this project context, entity represents a record with unique properties and id. It may include persons, records, subjects, etc.

Candidate Pair: An entity pair derived from the blocking technique. This pairs are obtained by rigorous filtering of entity comparisons. They are considered as potential pairs for matching.

Threshold: A numerical value defined based on the attribute. The similarity distance calculated between the entities is compared with it.

## 3.2 Existing Solutions:

There are many solutions defined to carry out the matching task in a parallel manner on a large scale. This is also called as Entity Matching over Big Data. Some of them are following:

### 3.2.1 Learning based approach:

In learning based approach, we use learning algorithms like Decision Tree, SVM, etc. while matching to determine the entities are match or not. The main problem for this approach is that initially training data should be provided for the learning algorithm. This data should be prepared manually by labelling a sample set of candidate pairs as match or non-match. This is a very hectic process as domain experts should carry out labelling and they need to analyze lot of attributes to determine whether two entities are match or non-match.

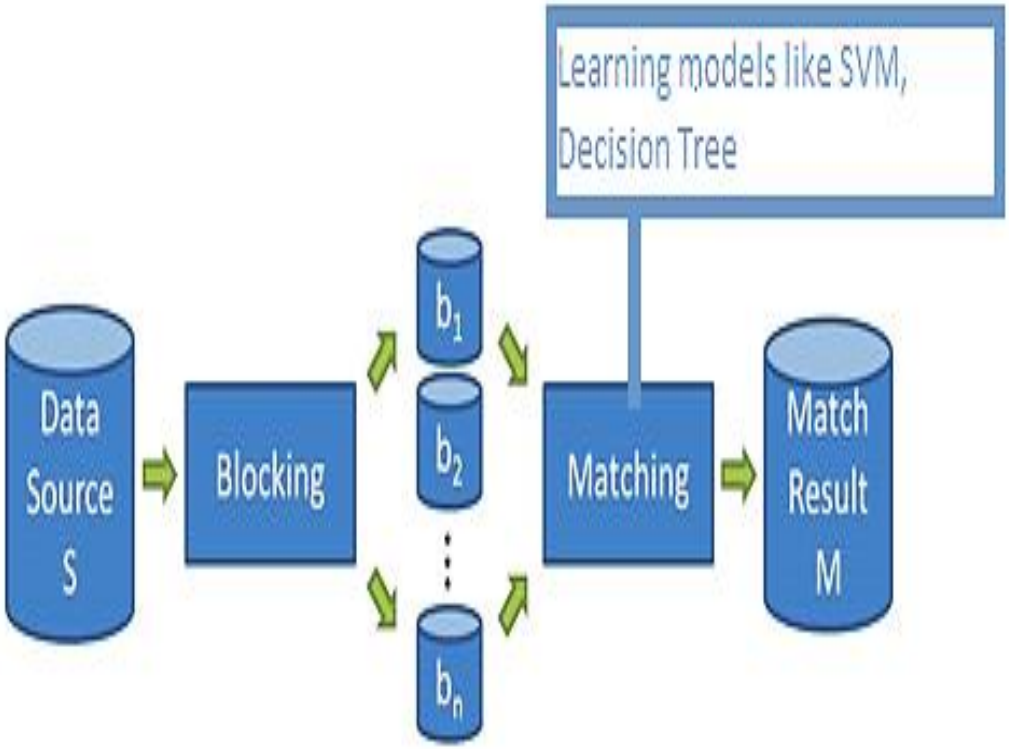


Figure 2: Learning Approach

There are some learning based frameworks like Active Atlas, MARLIN, Multiple Classifier System, Operator Trees, etc., which can be used to carry out the learning based strategy. This kind of approach is not scalable for large scale entity matching.

### 3.2.2 Non-Learning based approach:

In the non-learning based approach, we can use different distance functions to compute the similarity between the entities and classify them as match and non-match. For the sub-tasks until candidate calculation, the implementation is the same but differs only at instance, attribute and relationship level. In learning approaches, learning algorithms are used while matching and not used in non-learning approaches. The main advantage here is that there is no need to label the data manually. Here a lot of distance metrics like edit distance, Tf-Idf, levenstein distance, etc. can used to determine the similarity of the entities.

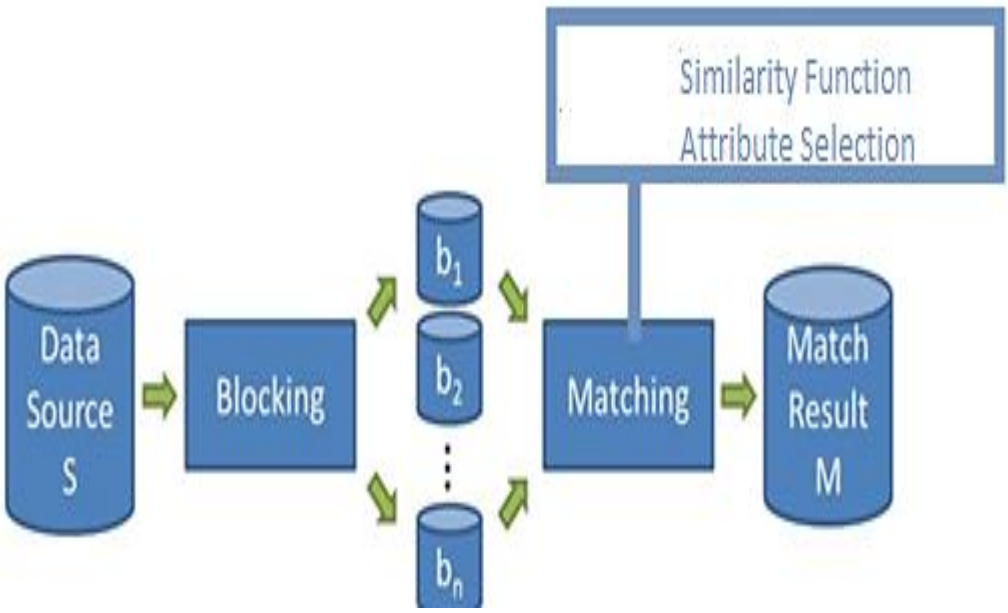


Figure 3: Non-Learning Approach

There are some frameworks without training like MOMA (Mapping based Object Matching), SERF (Stanford Entity Resolution Framework), etc. which can be used to classify the entities. The frameworks without training uses some distance computational measures and similarity functions to match the records. These frameworks can be implemented using big data technologies and they are scalable to big data. There are other approaches like Active learning, rule based matching, etc. that can be used to carry out the matching task.

### 3.3 Proposed Solution:

I have chosen to implement the non-learning approach. To scale the solution to larger data I have implemented it in Apache Spark. The following are different parts of the proposed solution.

#### 3.3.1 Hybrid Similarity Function:

The hybrid similarity function is a combination of different similarity functions, which decides the candidate pair as a match or non-match. Initially the candidate pairs obtained after the blocking strategy are fed as input to this function. I have chosen to implement a matcher function, which contains three similarity functions. They are Jaro similarity, edit-distance similarity and Jaccard similarity. The main goal is to run this hybrid similarity function parallely on different nodes to get better performance. This function calculates a similarity score for the candidate pairs taken as input and compares it with the threshold value. The pairs with score greater than the threshold value are considered as potential pairs.

The similarity score for the candidate pairs is calculated based on the individual scores obtained for the pairs from the three similarity functions. Certain weights are assigned to resulting scores from the similarity functions. Weighted average is taken from them

and if it is greater than the threshold value the candidate pair is considered as a potential match. I have followed other approaches like considering average of the individual distance scores calculated, considering highest value among calculated distance scores, etc. While merging distance scores calculated by three similarity functions.

### 3.3.2 R-Swoosh Algorithm:

To reduce the unnecessary comparisons occurring on different nodes we have used R-Swoosh algorithm. The following is the procedure of the algorithm:

i) Initially it takes two entities, compare them and if they match they are merged as one set. The next incoming entity is compared with only one of the entities in the previous set and if it matches, the entity is merged to the same set.

ii) If it is not matched, another set is created with the entity in the cases above. This process goes on continuously.

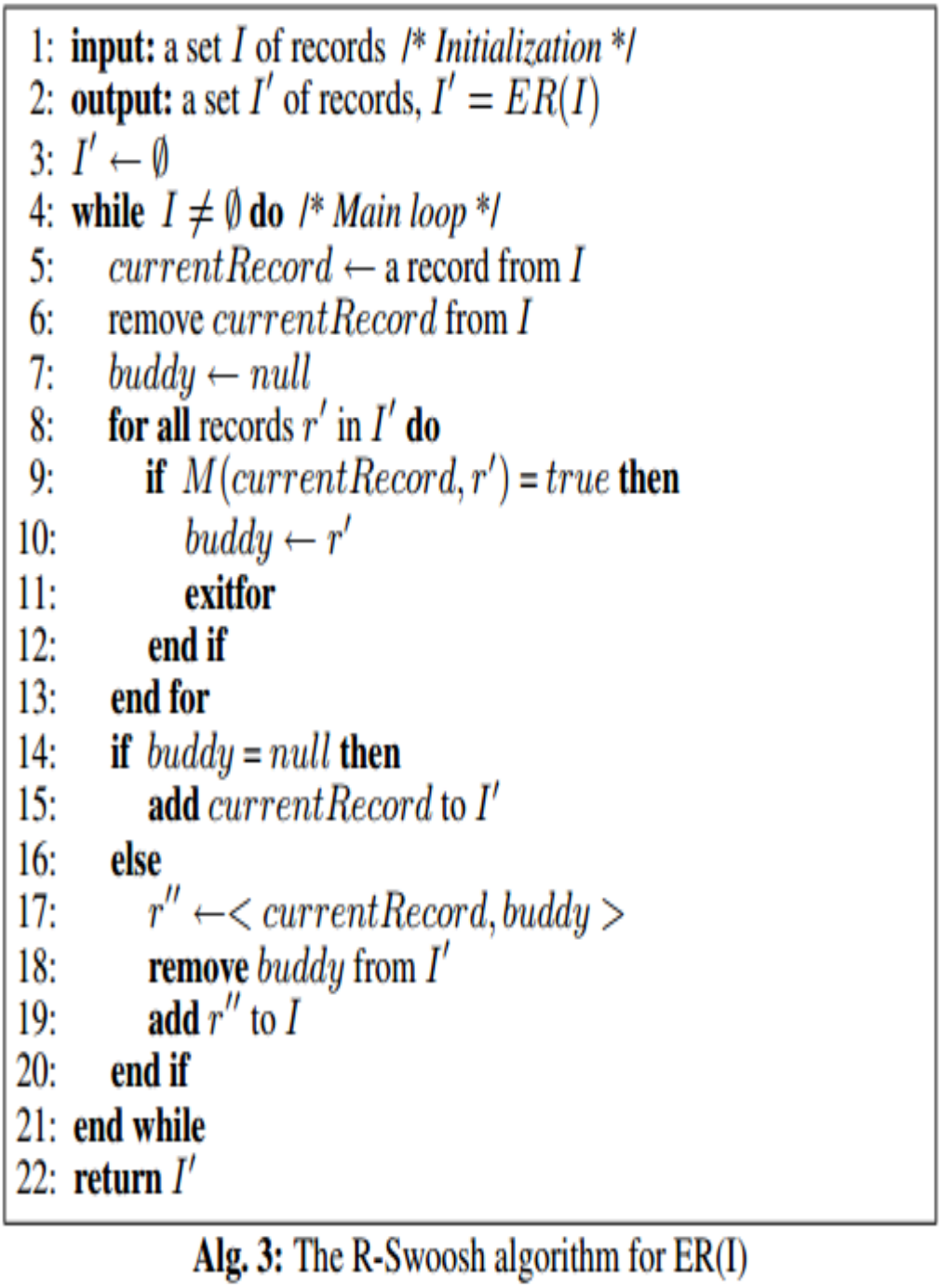


Figure 4: R-Swoosh Algorithm [3]

By using this algorithm, we can avoid unnecessary comparisons and the performance is increased. All the entity-to-entity comparisons are avoided and only selected

comparisons happen. The O (n2) time complexity is reduced. At the end the entities which are matching are derived.

### 3.3.3 Optimum Threshold value:

In General the threshold value used to evaluate the distance score calculated by the similarity function is given manually. It is guessed based on the type of attribute we are dealing. For instance, if the attribute is string and threshold value can be anywhere between [0, 1] for the similarity function chosen, but if the attribute is gender the threshold value should be exactly 1. Threshold value where the precision or recall will be maximum is termed as optimum threshold value. It is calculated based on the observation different similarity functions and thresholds have redundancy. [4]

The rule based matching discussed above is used to derive the optimum threshold value. Initially based on the attributes of the dataset attribute matching rules and record matching rules are derived. Based on the theorems defined in the paper “How Similar is Similar” we can calculate the optimum threshold value. Initially a sample set of entities from the dataset are taken and positive pairs and negative pairs are separated from them

**Theorem 1:**

*function* $F(\Psi, M, D)$. *Consider an* iRR *in* $\Phi$ *which contains an* iAR $\lambda^i : (a, \mathcal{F}, \Theta)$, *and two instances of* $\lambda^i$, $\lambda_1^e : (a, f, \theta_1)$ *and* $\lambda_2^e(a, f, \theta_2)$. *Suppose* $\Psi_1$ *is an* eRR *set,* $\Psi_2$ *is another* eRR *set transformed from* $\Psi_1$ *by replacing* $\lambda_1^e$ *in* $\Psi_1$ *with* $\lambda_2^e$. *If* $\theta_1 < \theta_2$ *and there is no positive example in* $\overline{M}_{\lambda_1^e} - \overline{M}_{\lambda_2^e}$, *we have* $F(\Psi_1, M, D) \leq F(\Psi_2, M, D)$.

Based on Theorem 1, we have an observation that a large number of eRRs can be pruned since they cannot provide a better objective value. For example, consider an iAR $\lambda_1^i : (name, \{f_e, f_g\}, [0, 1])$ in Figure 2. It has two similarity functions, edit similarity $f_e$ and gram-based similarity $f_g$. Table 1 shows $f_e(r[name], r'[name])$ and $f_g(r[name], r'[name])$ of each record pair $(r, r')$ in $E$. The positive examples are marked by the gray background color (e.g. $\text{RP}_{1,6}$). Consider two eARs $\lambda_1^e : (a, f_e, 0.6)$ and $\lambda_2^e : (a, f_e, 0.7)$, $\overline{M}_{\lambda_1^e} = \{\text{RP}_{1,3}, \text{RP}_{1,5}, \text{RP}_{1,6}, \text{RP}_{1,7}, \text{RP}_{2,5}, \text{RP}_{3,5}, \text{RP}_{3,6}, \text{RP}_{3,7}, \text{RP}_{5,7}, \text{RP}_{6,7}\}$, $\overline{M}_{\lambda_2^e} = \{\text{RP}_{1,3}, \text{RP}_{1,5}, \text{RP}_{1,6}, \text{RP}_{1,7}, \text{RP}_{2,5}, \text{RP}_{3,7}, \text{RP}_{6,7}\}$, $\overline{M}_{\lambda_1^e} - \overline{M}_{\lambda_2^e} = \{\text{RP}_{3,5}, \text{RP}_{3,6}, \text{RP}_{5,7}\}$. As there is no positive example in $\overline{M}_{\lambda_1^e} - \overline{M}_{\lambda_2^e}$, we can prune $\lambda_1^e$.

Figure 5: Theorem 1 [4]

According to this theorem, we will subtract entity pairs obtained from applying two threshold values to a similarity function. At least one pair from the resultant set should match with the positive set defined; otherwise we can prune the lower threshold value among them. By using this theorem, we can eliminate many threshold value options. This process is repeated continuously and at the end a set of threshold values are derived.

**Theorem 2:**

Given an iAR $\lambda^i : (a, \mathcal{F}, \Theta)$, there may be still many eARs of $\lambda^i$ in $\mathcal{P}(\lambda^i)$, we can remove some of them based on the following observation. Consider two eARs $\lambda_1^e : (a, f, \theta_1)$ and $\lambda_2^e : (a, f, \theta_2)$ in $\mathcal{P}(\lambda^i)$. Suppose $\theta_1 > \theta_2$, then $\overline{M}_{\lambda_1^e} \subset \overline{M}_{\lambda_2^e}$ (If $\overline{M}_{\lambda_1^e} = \overline{M}_{\lambda_2^e}$, $\lambda_2^e$ will be removed based on Theorem 1). Obviously, $\lambda_1^e$ and $\lambda_2^e$ generate the same record pairs $\overline{M}_{\lambda_1^e}$, and $\lambda_2^e$ can identify more record pairs $\overline{M}_{\lambda_2^e} - \overline{M}_{\lambda_1^e}$. If these pairs are all positive examples, that is $\overline{M}_{\lambda_2^e} - \overline{M}_{\lambda_1^e} \subseteq M$, then $\lambda_2^e$ will be better than $\lambda_1^e$. We say $\lambda_1^e$ is redundant w.r.t $\lambda_2^e$. The correctness is proved in Theorem 2.

THEOREM 2. *Given a* candidate eAR set $\mathcal{P}(\lambda^i)$ *and a set* $M$ *of positive examples,* $\lambda_1^e : (a, f, \theta_1) \in \mathcal{P}(\lambda^i)$ *is redundant w.r.t* $\lambda_2^e : (a, f, \theta_2) \in \mathcal{P}(\lambda^i)$ *if* $\overline{M}_{\lambda_2^e} - \overline{M}_{\lambda_1^e} \subseteq M$ *and* $\theta_1 > \theta_2$.

Figure 6: Theorem 2 [4]

This theorem is used to further prune threshold values obtained from the above set. According to this theorem, we will subtract entity pairs obtained from applying two threshold values to a similarity function from the set. At least one pair from the resultant set should not match with the positive set defined, otherwise we can prune the higher threshold value among them, as it is redundant. This process is repeated continuously and at the end a set of threshold values are derived.

**Theorem 3:**

Consider two eARs $\lambda_1^e : (a, f_1, \theta_1)$ and $\lambda_2^e : (a, f_2, \theta_2)$ in $\mathcal{P}(\lambda^i)$ where $f_1 \neq f_2$. We first consider a special case such that, for each record pair, $\lambda_1^e$ and $\lambda_2^e$ always generate the same record pairs, i.e. $\overline{M}_{\lambda_1^e} = \overline{M}_{\lambda_2^e}$. From the viewpoint of RR set, they have no difference and we can only keep one of them. To make this observation more general, we find that 1) for each positive pair $(r, r')$ (i.e. $(r, r') \in M$), if $\lambda_1^e$ takes this pair as similar, $\lambda_2^e$ will also takes it as similar, and 2) for each negative pair $(r, r')$ (i.e. $(r, r') \in D$), if $\lambda_1^e$ takes this pair as dissimilar, $\lambda_2^e$ will also takes it as dissimilar, then $\lambda_1^e$ is redundant w.r.t $\lambda_2^e$ since it can not lead to a better objective value than $\lambda_2^e$. The first statement implies $\overline{M}_{\lambda_1^e} \cap M \subseteq \overline{M}_{\lambda_2^e} \cap M$, and the second statement is equivalent to $\overline{M}_{\lambda_1^e} \cap D \supseteq \overline{M}_{\lambda_2^e} \cap D$. Theorem 3 shows the correctness.

THEOREM 3. *Given a* candidate eAR set $\mathcal{P}(\lambda^i)$, *a set* $M$ *of positive examples and a set* $D$ *of negative examples,* $\lambda_1^e : (a, f_1, \theta_1) \in \mathcal{P}(\lambda^i)$ *is redundant w.r.t* $\lambda_2^e : (a, f_2, \theta_2) \in \mathcal{P}(\lambda^i)$ *if* $\overline{M}_{\lambda_1^e} \cap M \subseteq \overline{M}_{\lambda_2^e} \cap M$ *and* $\overline{M}_{\lambda_1^e} \cap D \supseteq \overline{M}_{\lambda_2^e} \cap D$.

Figure 7: Theorem 3 [4]

This theorem is used to further prune threshold values obtained from the above set. According to this theorem, we will take the sets obtained from above step for two different similarity functions. In them, one threshold value from each set is taken and based on it the similar pairs and dissimilar pairs are calculated. If similar and dissimilar pairs in both cases are equal, then we can prune one threshold value from it.

By implementing these theorems in a sequential manner, we can narrow down the threshold values for a similarity function to a very small set. Now we will apply these threshold values on a sample set of data, calculate precision, and recall percentages of the result set. Whichever value has higher precision and accuracy will be considered as optimum threshold value for that similarity function.

# CHAPTER 4

# IMPLEMENTATION

## 4.1 Implementation Details:

For the implementation of the proposed solution, I have chosen Apache Spark framework. Initially I have experimented on smaller datasets and later done it on the larger ones and obtained satisfactory results. The following is the procedure for implementation.

The candidate pairs obtained from the blocking are considered as input. From them we need to extract the similar entities. The following is an input sample, which is fed to the spark implementation.

**Input:**

(2.0,Shum,SelinaWaiSheung)(2.0,Shum,SelinaWaiSheung)

(2.0,Shum,SelinaWaiSheung)(2.01,Pham,CuongHung)

(2.0,Shum,SelinaWaiSheung)(2.02,Kerali,HenryG.R.)

(2.01,Pham,CuongHung)(2.01,Pham,CuongHung)

(2.01,Pham,CuongHung)(2.02,Kerali,HenryG.R.)

(2.01,Pham,CuongHung)(2.02,Kerali,HenryG.R.)

(2.02,Kerali,HenryG.R.)(2.02,Kerali,HenryG.R.)

(2.02,Kerali,HenryG.R.)(2.02,Kerali,HenryG.R.)

(2.02,Kerali,HenryG.R.)(2.02,Kerali,HenryG.R.)

(2.02, Kerali, HenryG.R.)(2.03, Basu, Ananya)

(2.02, Kerali, HenryG.R.)(2.03, Basu, Ananya)

(2.02, Kerali, HenryG.R.)(2.04, Montes-Negret, Fernando)

The following is the detailed step-by-step explanation of implementation in spark.

**Pre-Processing Step:**

```
String testFile = "C:/Users/SAIRAM/Desktop/input.txt";
JavaSparkContext sc = new JavaSparkContext("local", "SimpleAPP");

JavaRDD<String> logData = sc.textFile(testFile).cache();

JavaRDD<String> processData = logData.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s){
        return Arrays.asList(s.replace(")(","#").split("#"));
    }
});

JavaRDD<String> cleanData = processData.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s){
        return Arrays.asList(s.replace("(", "").replace(")", ""));
    }
});
```

Figure 8: Spark code for pre-processing

Initially the input file is read through spark context and all the data in the text file is converted into RDD. The entity pairs are split into single entities and pre-processing like braces removal is applied as indicated in the above code. Then map step is applied on the resultant RDD from the pre-processing step.

**Map Step:**

```
JavaPairRDD<Double, String> groupData = cleanData.mapToPair(new PairFunction<String, Double, String>() {
    public Tuple2<Double, String> call(String s) {
    int count = 0;
    Double index = 0.0;
    StringBuilder sb = new StringBuilder();
    String[] items = s.split(",");
    for(String x : items){
        if(count == 0){
            count++;
            index = Double.parseDouble(x);
        }
        else{
            sb.append(x);
            sb.append(' ');
        }
    }
            return new Tuple2<Double, String>(index, sb.toString());
          }
    });
```

Figure 9: Spark code for Map Step

The map step is performed on the resultant RDD from pre-processing step. The entities are mapped with the unique blocking key provided initially from blocking and the string data is concatenated together using stringbuilder. The string data is concatenated to make it easy for comparison while applying similarity functions. Entities with same blocking key are mapped together. The blocking unique key plays a key role here as it brings the similar entities together which avoids unnecessary comparisons between dissimilar entities. The output for it would be in this manner

**Format:**

2.01: PhamCuongHung

2.01: PhamCuongHung

2.01: PhamCuongHung

2.02: KeraliHenryG.R.

2.02: KeraliHenryG.R.

2.05: ChallaKrishna

2.05: ChallaKrishna

2.05: ChallaKrishna

**Reduce Step:**

```
JavaPairRDD<Double,Iterable<String>> groupsOfData = groupData.groupByKey();

File outputFile = new File("C:\\Users\\SAIRAM\\Desktop\\clusters.txt");
FileWriter fw = new FileWriter(outputFile);

int clusterCount = 0;
for(Tuple2<Double, Iterable<String>> x : groupsOfData.collect()){
    fw.write(Main.stringDistance(x._2,clusterCount));
    clusterCount++;
}
```

Figure 10: Spark code to Group similar entities together

```
public static String stringDistance(Iterable<String> list,int clusterCount){
    Levenshtein l = new Levenshtein();
    JaroWinkler jw = new JaroWinkler();
    LongestCommonSubsequence lc = new LongestCommonSubsequence();
    Map<Integer,List<String>> clusters = new HashMap<Integer,List<String>>();
    int index = 0;
    for(String x : list){
        if(index == 0){
            List<String> arrayList = new ArrayList<String>();
            arrayList.add(x);
            clusters.put(index,arrayList);
            index++;
        }
        else{
            for(int indexer = 0; indexer < clusters.size(); indexer++){
                List<String> arrayList = new ArrayList<String>();
                arrayList = clusters.get(indexer);
                try{
                    if((((l.distance(x, arrayList.get(0)))+(lc.distance(x, arrayList.get(0)))+
                            (jw.distance(x, arrayList.get(0))-1))/3)<0.3){
                        arrayList.add(x);
                        clusters.remove(indexer);
                        clusters.put(indexer,arrayList);
                        break;
                    }
                }catch(NullPointerException e){
                    e.printStackTrace();
                }
            }
        }
    }
```

Figure 11: Spark code for implementation of Hybrid matcher function and R-Swoosh

Finally by applying the reduce step through groupofdata.collect() function the similar entities are grouped together and written to the output file. Based on the Blocking unique key the entities are grouped together. While doing so the mixed matcher function along with the R-Swoosh algorithm is applied by calling the stringDistance function, which determines the similar entities. The following is the final output obtained from the candidate pairs taken initially

**Output Format:**

CLUSTERS: [Verma Niraj, Verma Niraj, Verma Niraj, Verma Niraj, Verma Niraj, Verma Niraj]

CLUSTERS: [Berryman SueEllen, Berryman SueEllen, Berryman SueEllen, Berryman SueEllen, Berryman SueEllen, Berryman SueEllen]

CLUSTERS: [Ouedraogo IsmaelS., Ouedraogo IsmaelS., Ouedraogo IsmaelS., Ouedraogo IsmaelS., Ouedraogo IsmaelS., Ouedraogo IsmaelS.]

CLUSTERS: [Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian, Diou Christian]

CLUSTERS: [Kubota Keiko, Kubota Keiko, Kubota Keiko, Kubota Keiko, Kubota Keiko, Kubota Keiko]

CLUSTERS: [Mr.Constant Amouali, Mr.Constant Amouali, Mr.Constant Amouali, Mr.Constant Amouali, Mr.Constant Amouali, Mr.Constant Amouali]

CLUSTERS: [Prevost YvesAndre, Prevost YvesAndre, Prevost YvesAndre, Prevost YvesAndre, Prevost YvesAndre, Prevost YvesAndre]

CLUSTERS: [Mr.AdelinoCastelo David, Mr.AdelinoCastelo David, Mr.AdelinoCastelo David, Mr.AdelinoCastelo David, Mr.AdelinoCastelo David, Mr.AdelinoCastelo David]

**Optimization Implementation:**

The main issue is the threshold value, which is provided manually for the hybrid similarity function. Any approcah to obtain an optimum threshold value for the hybrid matcher function can increase the precision .By implementing the theorems mentioned form the paper, “How Similar is Similar” sequentially the optimum threshold value is found and used for the similarity functions in the hybrid matcher function. The following are the optimum threshold values for the similarity functions I have used. A sample entity set is initially considered for determining the optimum threshold value. Manually we should determine the similar entities and dissimilar entities and group them as positive and negative sets.

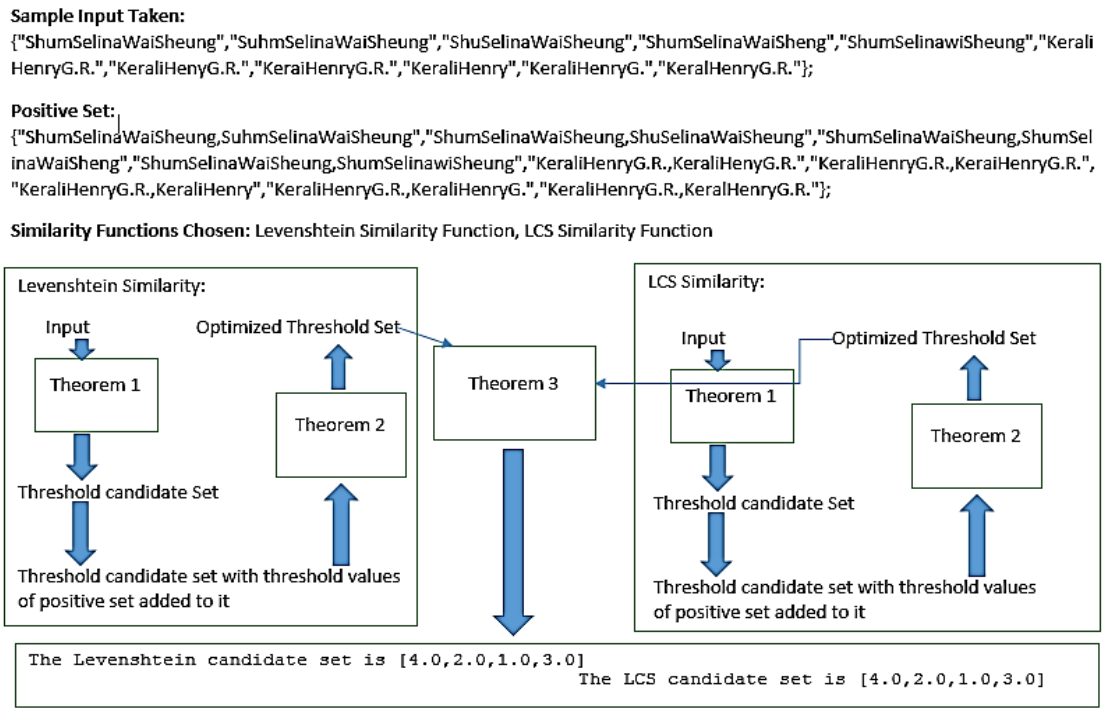


Figure 12: Theorems implementation on Levenshtein and LCS similarity functions

**Threshold Output:**

By applying the theorems on the sample set using Levenshtein and LCS similarity functions I have obtained the following candidate threshold sets. Apply these threshold values on a sample set and calculate precision. The threshold value with highest precision is the optimum threshold value. For Levenshtein similarity 2.0 gives better result and for Least common subsequence similarity 3.0 gives better result. Based on the nature of the dataset maximum, minimum and average value of the candidate threshold set in the baseline to improve the result.

```
The Levenshtein candidate set is [4.0, 2.0, 1.0, 3.0]

The Lcs candidate set is [4.0, 2.0, 1.0, 3.0]
```

I have done the experimentation in two modes. One is on local cluster mode and the other is on VM cluster mode

### 4.2 Spark Cluster Setup:

In the Virtual Machine mode, I used Ubuntu 1.4 cluster with four nodes each with 2GB of RAM and 80GB of virtual storage. The following are the steps to setup the cluster.

1. Initially install Java SDK on the Virtual Machine.
2. All the Virtual Machines are provided with remote access inside the cluster.
3. Install Hadoop Distributed File System using Hadoop and configure the libraries.
4. Download and install the Spark framework.
5. After installation start the Spark cluster.
6. Open chrome browser and type Master IP: 8080 ports to validate the Spark UI context.

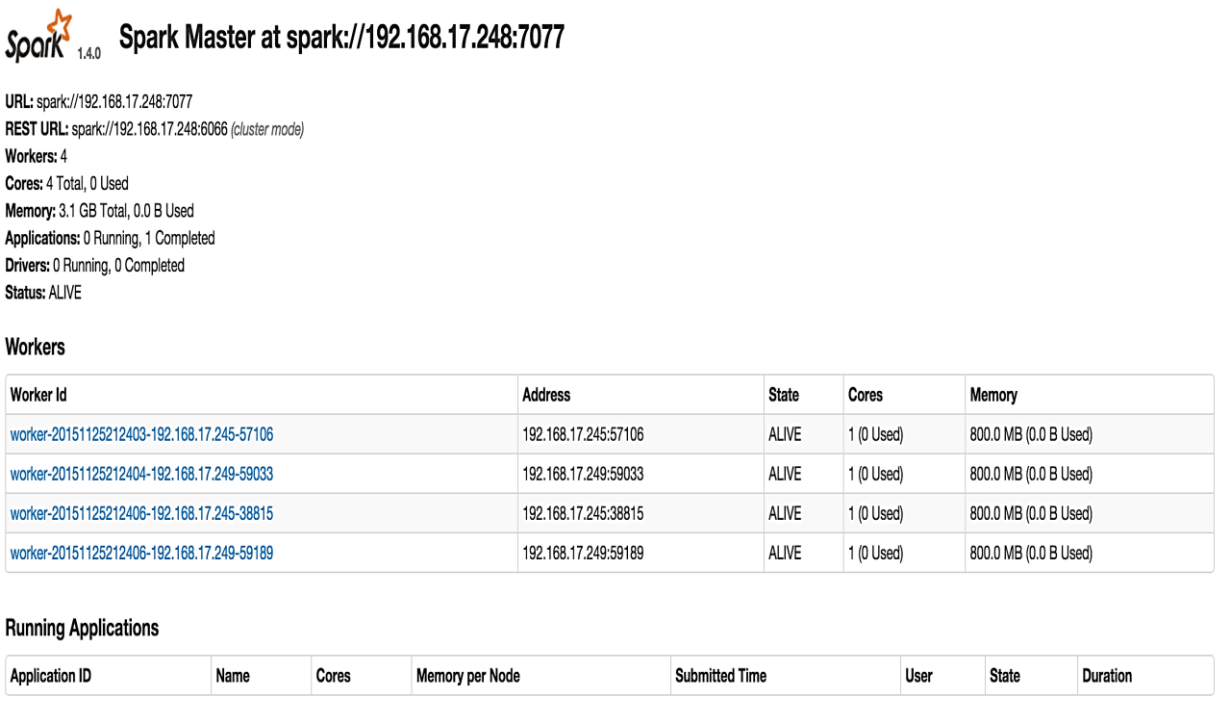


Figure 13: Spark Cluster UI Context Link

### 4.3 Sample Code:

Sample code of implementation of hybrid matcher function with R-Swoosh algorithm

```
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.PairFunction;
import info.debatty.java.stringsimilarity.NormalizedLevenshtein;
import scala.Tuple2;

public class TestClass {
    public static void stringDistance(Iterable<String> list){
        NormalizedLevenshtein l = new NormalizedLevenshtein();
        Map<Integer,List<String>> clusters = new HashMap<Integer,List<String>>();
        int index = 0;
        for(String x : list){
            if(index == 0){
                List<String> arrayList = new ArrayList<String>();
                arrayList.add(x);
                clusters.put(index,arrayList);
                index++;
            }
            else{
                for(int indexer = 0; indexer < clusters.size(); indexer++){
                    List<String> arrayList = new ArrayList<String>();
                    arrayList = clusters.get(indexer);
                    try{
                        if(l.distance(x, arrayList.get(0))<0.3){
                            arrayList.add(x);
                            clusters.remove(indexer);
                            clusters.put(indexer,arrayList);
                            break;
                        }
```

Figure 14: Sample code for Baseline -1

```
                    }
                }catch(NullPointerException e){
                    e.printStackTrace();
                }
            }
        }
        int clusterCount = 0;
        for(List<String> x : clusters.values()){
            System.out.println(String.valueOf(clusterCount) + "\t" + x);
            clusterCount++;
        }
    }
    @SuppressWarnings("serial")
    public static void main(String[] args){
        String testFile = "C:/Users/siddarth/Desktop/Vimal/IIA Dataset input.txt";
        JavaSparkContext sc = new JavaSparkContext("local", "SimpleAPP");

        JavaRDD<String> logData = sc.textFile(testFile).cache();

        JavaRDD<String> processData = logData.flatMap(new FlatMapFunction<String, String>() {
            public Iterable<String> call(String s){
                return Arrays.asList(s.replace(")(","#").split("#"));
            }
        });
        JavaRDD<String> cleanData = processData.flatMap(new FlatMapFunction<String, String>() {
            public Iterable<String> call(String s){
                return Arrays.asList(s.replace("(", "").replace(")", ""));
            }
        });
                String[] items = s.split(",");
                for(String x : items){
                    if(count == 0){
                        count++;
                        index = Double.parseDouble(x);
                    }
                    else{
```

Figure 15: Sample code for Baseline -2

```
                String[] items = s.split(",");
                for(String x : items){
                    if(count == 0){
                        count++;
                        index = Double.parseDouble(x);
                    }
                    else{
                        sb.append(x);
                    }
                }
                return new Tuple2<Double, String>(index, sb.toString());
            }
        });
        JavaPairRDD<Double,Iterable<String>> groupsOfData = groupData.groupByKey();
        for(Tuple2<Double, Iterable<String>> x : groupsOfData.collect()){
            TestClass.stringDistance(x._2);
        }
        sc.close();
    }
```

Figure 16: Sample code for Baseline -3

Sample Code for implementation of Optimum Threshold value

```
import info.debatty.java.stringsimilarity.*;

public class threshold {

    public static final Levenshtein distanceUtil = new Levenshtein();
    public static final LongestCommonSubsequence lcsDistanceUtil = new LongestCommonSubsequence();

    public static void main(String args[]) {

        // Getting Positive Set
        HashSet<String> positiveSet = getPositiveSet();

        /*********************************************************************************************
         * Using Levenshtein for the first Iteration. Computing candidate set
         *********************************************************************************************/
        // Calculating distance between pairs of dataSet.
        ArrayList<NamePair> levenshteinNamePairs = calculatePairsDistance(getDataSet(), "Levenshtein");
        // Storing the distance -> namePairs mapping
        HashMap<Double, ArrayList<String>> levenshteinDistanceNamePairMapping = getDistanceNamePairMapping(
                levenshteinNamePairs);
        // Getting distance between the pairs in positive set.
        ArrayList<NamePair> levenshteinPositveSetNamePairs = calculatePositivePairsDistance(positiveSet, "Levenshtein
        // Computing the candidate set.
        HashSet<Double> levenshteinCandidateSet = getCandidateSet(positiveSet, levenshteinDistanceNamePairMapping);
        // Corollary-1
        for (int i = 0; i < levenshteinPositveSetNamePairs.size(); i++) {
            levenshteinCandidateSet.add(levenshteinPositveSetNamePairs.get(i).getDistance());
        }
        // Computing the final candidate set.
        HashSet<Double> finalLevenshteinCandidateSet = getFinalCandidateSet(positiveSet,
                levenshteinDistanceNamePairMapping, levenshteinCandidateSet);

        /*********************************************************************************************
         * Using LCS for the first Iteration. Computing candidate set
```

Figure 17: Sample code for Optimization -1

```
/*********************************************************************************************
 * Comparing the Levenshtein candidate set and LCS candidate set and
 * producing the final candidate set
 *********************************************************************************************/
for (int i = 10; i >= 0; i--) {
    ArrayList<String> namePairHighList = new ArrayList<String>();
    for (int j = i; j >= 0; j--) {
        Double highDistance = new Double(j);
        if (finalLevenshteinCandidateSet.contains(highDistance)
                && levenshteinDistanceNamePairMapping.containsKey(highDistance)) {
            namePairHighList.addAll(levenshteinDistanceNamePairMapping.get(highDistance));
        }
    }
    ArrayList<String> namePairLowList = new ArrayList<String>();
    for (int k = i - 1; k >= 0; k--) {
        Double lowDistance = new Double(k);
        if (finalLcsCandidateSet.contains(lowDistance) && lcsDistanceNamePairMapping.containsKey(lowDistance)) {
            namePairLowList.addAll(lcsDistanceNamePairMapping.get(lowDistance));
        }
    }
    if (namePairHighList.containsAll(namePairLowList) && namePairHighList.size() == namePairLowList.size()) {
        levenshteinCandidateSet.remove(new Double(i));
    }
}
for (int i = 10; i >= 0; i--) {
    ArrayList<String> namePairHighList = new ArrayList<String>();
    for (int j = i; j >= 0; j--) {
        Double highDistance = new Double(j);
        if (finalLcsCandidateSet.contains(highDistance)
                && lcsDistanceNamePairMapping.containsKey(highDistance)) {
            namePairHighList.addAll(lcsDistanceNamePairMapping.get(highDistance));
        }
    }
    ArrayList<String> namePairLowList = new ArrayList<String>();
    for (int k = i - 1; k >= 0; k--) {
        Double lowDistance = new Double(k);
```

Figure 18: Sample code for Optimization -2

```
            if (finalLevenshteinCandidateSet.contains(lowDistance) && levenshteinDistanceNamePairMapping.containsKey(lowDistance)) {
                namePairLowList.addAll(levenshteinDistanceNamePairMapping.get(lowDistance));
            }
        }
        if (namePairHighList.containsAll(namePairLowList) && namePairHighList.size() == namePairLowList.size()) {
            lcsCandidateSet.remove(new Double(i));
        }
    }
    System.out.println("The Levenshtein candidate set is " + finalLevenshteinCandidateSet);
    System.out.println("The Lcs candidate set is " + finalLcsCandidateSet);
}

/*
 * Computes the distance between all possible pairs.
 */
public static ArrayList<NamePair> calculatePairsDistance(String[] dataSet, String algorithm) {
    if (algorithm.equals("Levenshtein")) {
        ArrayList<NamePair> namePairs = new ArrayList<NamePair>();
        for (int i = 0; i < dataSet.length; i++) {
            for (int j = i + 1; j < dataSet.length; j++) {
                String name = dataSet[i] + "," + dataSet[j];
                double distance = distanceUtil.distance(dataSet[i], dataSet[j]);
                namePairs.add(new NamePair(name, distance));
            }
        }
        return namePairs;
    } else {
        ArrayList<NamePair> namePairs = new ArrayList<NamePair>();
        for (int i = 0; i < dataSet.length; i++) {
            for (int j = i + 1; j < dataSet.length; j++) {
                String name = dataSet[i] + "," + dataSet[j];
                double distance = lcsDistanceUtil.distance(dataSet[i], dataSet[j]);
                namePairs.add(new NamePair(name, distance));
            }
        }
        return namePairs;
    }
```

Figure 19: Sample code for Optimization -3

```
/*
 * Computes the distance between the pairs of string in Positive Set.
 */
public static ArrayList<NamePair> calculatePositivePairsDistance(HashSet<String> positiveSet, String algorithm) {
    if (algorithm.equals("Levenshtein")) {
        ArrayList<NamePair> namePairs = new ArrayList<NamePair>();
        Iterator iterator = positiveSet.iterator();
        while (iterator.hasNext()) {
            String name = (String) iterator.next();
            String firstName = name.split(",")[0];
            String secondName = name.split(",")[1];
            namePairs.add(new NamePair(name, distanceUtil.distance(firstName, secondName)));
        }
        return namePairs;
    } else {
        ArrayList<NamePair> namePairs = new ArrayList<NamePair>();
        Iterator iterator = positiveSet.iterator();
        while (iterator.hasNext()) {
            String name = (String) iterator.next();
            String firstName = name.split(",")[0];
            String secondName = name.split(",")[1];
            namePairs.add(new NamePair(name, lcsDistanceUtil.distance(firstName, secondName)));
        }
        return namePairs;
    }
}

// Function to fetch the data Set
public static String[] getDataSet() {
    String[] input = {"ShumSelinaWaiSheung","SuhmSelinaWaiSheung","ShuSelinaWaiSheung","ShumSelinaWaiSheng",
            "ShumSelinawiSheung","KeraliHenryG.R.","KeraliHenyG.R.","KeraiHenryG.R.","KeraliHenry","KeraliHenryG.",
            "KeralHenryG.R."};
    return input;
}
```

Figure 20: Sample code for Optimization -4

### 4.4 Execution Report:

Medicare's Helpful Contacts Dataset is used for the experimentation purpose. I have done the experimentation in two modes. One is on local cluster mode and the other is on VM cluster mode. In the Virtual Machine mode, I used Ubuntu 1.4 cluster with four nodes each with 2GB of RAM and 80GB of virtual storage.

**SparkConf().setMaster("spark://192.168.17.248:7077").setAppName("VimalApp");**

In local mode, I have use Local mode with four nodes and driver memory of 3 GB.

**SparkConf().setMaster("local[4]").setAppName("VimalApp");**

Spark Distribution:

To perform Entity Matching I have used the same data as source for comparison, for first round of partitioning, I was dependent on the default partition index of the spark, which divides the data into default partition size 64 mb. It is same as Hadoop.

**JavaRDD<String> logData = sc.textFile(testFile).cache();**

The following is the runtime performance for the experiments I have conducted.

Data Loading: It initially cache the data into a RDD

**JavaRDD<String> logData = sc.textFile(testFile).cache();**

For every job, the following are the time taken recorded in seconds. Below all-time comparison is based on more than 03, 35,433 entities

**Time for job initiation**: Load data stage 0 time of execution is 91 seconds

**Time for map task**: Map Task has taken 117 seconds for pairs processing and key calculation

**Time for Reduce task**: The reduce task has taken 187 seconds. In it the pairs have

**Time for writing data back to disk**: Spark program had 1 time writing the data into disk, for Pairs after all comparisons, which is maximum time

The following is the sample Workload distribution among nodes: Spark Default partition and workload distributions among four nodes

15/10/19 23:17:29 INFO MemoryStore: ensureFreeSpace(189127) called with curMem=1260025, maxMem=56973721

15/10/19 23:17:29 INFO MemoryStore: ensureFreeSpace(396537) called with curMem=1496909, maxMem=56973721

15/10/19 23:17:29 INFO MemoryStore: ensureFreeSpace(242217) called with curMem=1942717, maxMem=56973721

15/10/19 23:17:29 INFO MemoryStore: ensureFreeSpace(437681) called with curMem=2144943, maxMem=56973721

# CHAPTER 5

# PERFORMANCE EVALUATION

I have evaluated the hybrid matcher function using three critical factors:

1. Matching result in terms of Precision
2. Running time with and without the comparison reduction algorithm
3. Configuring number of maps and reduce tasks, number of available nodes in clusters

## 5.1 Precision:

Precision is the ratio of the number of relevant match pairs retrieved to the total number of irrelevant and relevant match pairs retrieved. To validate the precision for this data I have put a counter in the reduce step which increments whenever we find a match. The ratio of the counter to the total number of pairs I have initially fed to the matcher function gives us the precision.

I have calculated the precision for the dataset with the hybrid matcher function and each of the similarity function in the matcher functionally individually. I have found that using a hybrid matcher function increased the precision to an extent.

The following graph shows us the precision values against similarity functions I have used on the dataset. It shows the precision have increased for the hybrid matcher function we have used

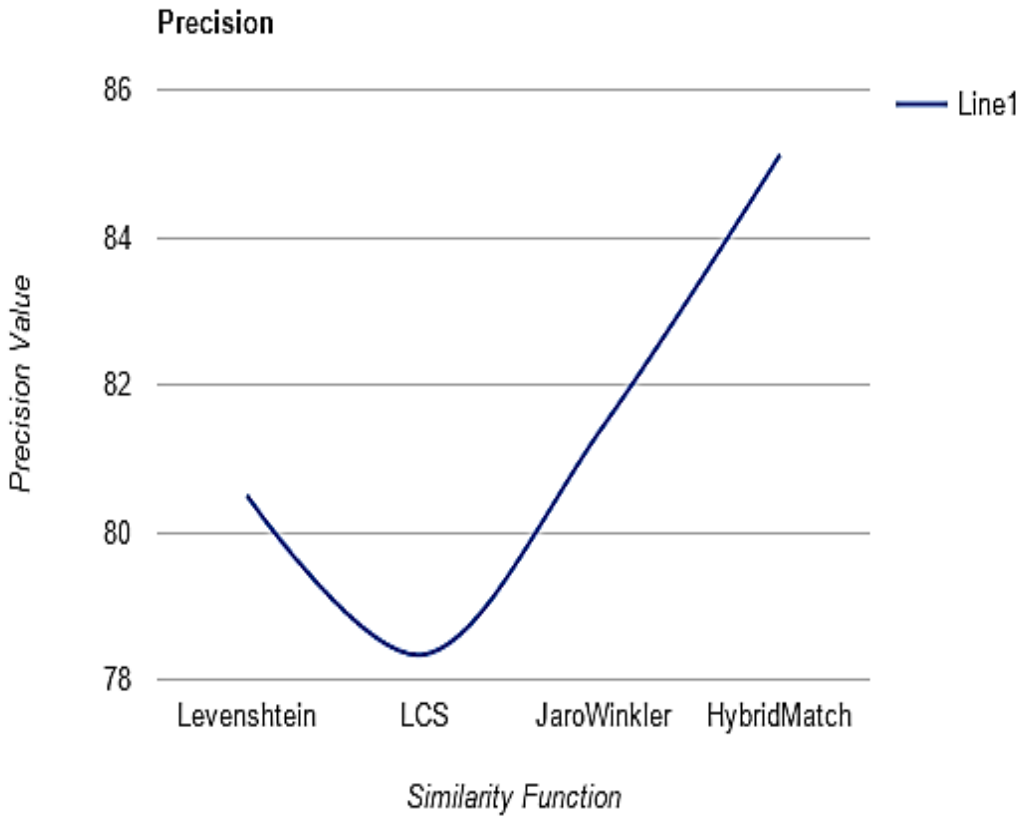


Figure 21: Graph for Precision Vs Similarity Function

To derive the threshold value for the hybrid matcher function there are many approaches like MIN-MAX approach, weighted approach, average approach, etc. I have tried all of these approaches and observed that weighted approach yields better results. Different weights are assigned to the calculated similarity scores of the similarity functions in the hybrid matcher function. These weights are summed to a score and if it is greater than the threshold value for the hybrid matcher function then the pair is considered as match. For example, '0.9' is considered as the threshold value and if all the weights of the similarity functions calculated is greater than 0.9 then the entity pair is considered as match otherwise as non-match.

### 5.2 Running Time:

I have calculated the running time for the hybrid matcher function without the comparison reduction algorithm and with it. I have observed that the running time with the algorithm is lesser than the regular approach. By using R-swoosh algorithm, we can reduce the number of comparisons.

The issue with the hybrid matcher function is that we will be giving the threshold value manually and based on it the entities are classified as match or non-match. By implementing the optimum threshold value for the hybrid matcher function, we can see an increase in the precision.

### 5.3 Increase in number of Reduce Tasks and nodes:

By increasing the number of reducers, the execution time decreases as the distribution of the reduce task has increased.

To increase the precision of the resultant pairs we can optimize the baseline solution by implementing the “Optimum threshold value” concept mentioned above. I have implemented the approach and used the optimum threshold values obtained for different similarity functions obtained from it and observed that the precision of resultant set have increased.

**CHAPTER 6**

# ADVANTAGES

1. The precision of the entity matching can be increased by implementing the hybrid matcher function with optimum threshold value.
2. The unnecessary comparisons while matching can be reduced by using R-Swoosh algorithm along with hybrid matcher function
3. Higher precision is obtained by implementing the optimum threshold value, which eliminates threshold redundancy and similarity function redundancy.
4. Implementation using spark reduce the writing multiple times into disc to single entry to disc.

**CHAPTER 7**

# Conclusion

The proposed hybrid matcher function approach with R-swoosh algorithm for parallelizing matching task of Entity resolution using the widely available MapReduce framework effectively distributes the workload and returns the entity pairs, which are a match. Our evaluation in a real cluster environment with one master and two worker nodes using real-world data demonstrated that approach works effectively and scale with available number of nodes. The optimized hybrid matcher function approach using Optimum threshold value improved the Precision.

## LIST OF REFERENCES

[1] http://spark.apache.org/docs/latest/cluster-overview.html. [Accessed: 01- Dec- 2015].

[2] Vibhor Rastogi, Nilesh Dalvi and Minos Garofalakis∗, "Large-Scale Collective Entity Matching"

[3] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang and Jennifer Widom, "Swoosh: a generic approach to entity resolution"

[4] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu and Jianhua Feng, "Entity Matching: How Similar Is Similar"

[5] https://github.com/tdebatty/java-string-similarity [Accessed: 23- Nov- 2015].

[6] https://en.wikipedia.org/wiki/Jaccard_index [Accessed: 23- Nov- 2015].

[7] https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance [Accessed: 23- Nov- 2015].

[8] https://en.wikipedia.org/wiki/Edit_distance [Accessed: 23- Nov- 2015].