Scalable Construction of a Large IsA-Knowledge Base from Heterogeneous Web Data

Muhammad Salman Sadaqat

Technische Universität Dresden
Department of Computer Science
Database Technology Group

October 14, 2014

supervisor: Dr.-Ing. Dirk Habich
professor: Prof. Dr.-Ing. Wolfgang Lehner

Declaration

Herewith I declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher education, except where due acknowledgment has been made in the text.

Dresden, October 14, 2014

Muhammad Salman Sadaqat

Acknowledgements

I am glad to acknowledge those who have supported me during the time I've spent for this thesis.

I would like to extend my heartfelt gratitude to my supervisor, Dr.-Ing. Dirk Habich along with Katrin Braunschweig and Julian Eberius for guiding me throughout this project and giving me invaluable advice at times when I needed it the most.

I would like to thank all the authors whose work was helpful during this thesis. I would also like to thank the open source fraternity who are always there to help others and take out the time to respond to even the most basic queries.

And last, but certainly not the least, to my parents who made me who I am today!

Contents

1 Introduction 7
- 1.1 Background . . . 7
- 1.2 Objective . . . 8
- 1.3 Motivation . . . 8
- 1.4 Problem Statement . . . 9

2 Foundations 11
- 2.1 Knowledge Base . . . 11
  - 2.1.1 Knowledge Base Creation . . . 11
- 2.2 Taxonomy . . . 12
- 2.3 Map-Reduce . . . 13
  - 2.3.1 Programming Model . . . 13
  - 2.3.2 Execution Structure . . . 15

3 Set Similarity Join Algorithms and Architecture 17
- 3.1 Similarity Algorithms . . . 17
  - 3.1.1 Jaccard Similarity Coefficient . . . 18
  - 3.1.2 Dice Coefficient . . . 19
  - 3.1.3 Comparison between Jaccard and Dice coefficients . . . 21
- 3.2 Set Similarity Joins . . . 22
  - 3.2.1 Background Overview . . . 22
  - 3.2.2 Similarity Cases . . . 23
  - 3.2.3 Basic Set Similarity Algorithm . . . 26
- 3.3 Set Similarity Joins with Map-Reduce . . . 28
  - 3.3.1 Set Similarity Identification . . . 28
  - 3.3.2 Spelling Difference Identification . . . 31
  - 3.3.3 Synonym Identification . . . 35
- 3.4 Architectural Overview . . . 41

4 Evaluation and Analysis **42**

4.1 Threshold for Set Similarity Joins 42
4.2 Threshold for String Similarity 44
4.3 Environment Details 45
4.4 Set Similarity Joins 46
4.5 Spelling Difference Identification 49
4.6 Synonym Identification 51
4.7 Analysis 54

5 Related Work **55**

5.1 Efficient Parallel Set-similarity Joins Using MapReduce 55
5.2 Efficient Similarity Joins for Near Duplicate Detection 57
5.3 Efficient Exact Set Similarity Joins 57
5.4 MassJoin: A mapreduce-based method for scalable string similarity joins 58
5.5 Efficient Set Joins on Similarity Predicates 58
5.6 Probase: A Probabilistic Taxonomy for Text Understanding . . 59

6 Conclusion **60**

References **61**

Index **63**

List of Figures

2.1 A Simple Taxonomy. 12
2.2 Execution Overview of Map-Reduce 15

3.1 Jaccard coefficient for sets A and B 18
3.2 Sets A and B with Jaccard index 3/11 18
3.3 Jaccard distance for sets A and B 19
3.4 Sørensen index for samples A and B 19
3.5 String Similarity Measure 20
3.6 Execution Overview of Map-Reduce 29
3.7 First Map-Reduce Job for Spelling Difference Identification . . . 32
3.8 Second Map-Reduce Job for Spelling Difference Identification . . 34
3.9 First Map-Reduce Job for Synonym Identification 36
3.10 Second Map-Reduce Job for Synonym Identification 38
3.11 Architectural Overview 41

4.1 System Specification Details 45
4.2 Cluster Details 45
4.3 Overview of Experiments for Set Similarity Joins 46
4.4 No. of Records processed with respect to time 47
4.5 Efficiency Overview 48
4.6 Experiments with Duplicate Input Records 49
4.7 Overview of Experiments for Spelling Difference Identification . 49
4.8 Efficiency Overview 50
4.9 Overview of Records Reduced 51
4.10 Overview of Experiments for Synonym Identification 52
4.11 Efficiency Overview 52
4.12 Overview of Records 53

5.1 Three phase approach for Set Similarity Joins 56

List of Tables

3.1 Step wise execution of Reduce Function 35
3.2 Reducer 40

4.1 Experiments for Threshold Values 43
4.2 Threshold Values for Set Similarity Identification 44
4.3 Threshold Analysis for String Similarity 44

Chapter 1

Introduction

This chapter provides a basic introduction to the thesis. The background and scope of this thesis, the objective, the motivation and the problem statement are discussed in detail.

1.1 Background

Processes involving information retrieval, data analysis and reasoning play an important role in many applications. Therefore, usage of such a centralized information system has become very common. It requires structuring of information in a hierarchical order and one way to achieve this is by creating a knowledge base or taxonomy (explained in detail in the proceeding chapter).

Taxonomies can be constructed through automatic information extraction techniques and the extracted data can be integrated and then formalized in the form of large trees or graph structures [10]. There are two main processes involved in the construction of taxonomy i.e. Data Extraction and Integration.

The extraction process normally starts by selecting a text corpus as an input and then information extraction methods are used to extract data and relationships between them. Once the extraction process is completed, the data undergoes through an integration process where the duplicate data sets are merged together in order to construct a taxonomy with unique data sets.

The whole process can take considerable amount of time and resources. Therefore, highly efficient techniques are required for information extraction from large data sources. Furthermore, data integration phase also needs the same level of efficiency as well. To cope with the requirements, distributed data processing techniques are used in making these processes more efficient.

1.2 Objective

The objective of this thesis is to formulate an efficient process for the construction of a large and scalable Is-A knowledge base from heterogeneous data sources. The efficiency of the process not only depends on the information extraction phase but also on 'how efficiently the extracted information is integrated afterwards'. Therefore the objective of this thesis is to build an efficient information integration process which results in a large and scalable Is-A knowledge base.

1.3 Motivation

The dataset on the corpus consists of hundreds of Tera bytes. The size of Common Crawl is more than 260 TB containing over 4 billion web pages. This enormous amount of data needs to be processed in an efficient way. Therefore, distributed data processing techniques like Map-Reduce have been widely used for information extraction from the corpus. They have enabled the processing of extensively large sizes of data in an efficient way in terms of time and resources.

During the extraction process, the data is extracted in the form of sentences having keywords: "such as", "like", "including", "especially" and others as well which identify 'Is-A relationship' between the concepts within a sentence. Once the extraction phase is completed, the next phase is to integrate the information extracted (Is-A pairs). The motivation for this thesis is to create an efficient process on how to merge the similar data sets using Map-Reduce framework.

1.4 Problem Statement

Taxonomy construction is a laborious process and large a lot of researchers have contributed on how this process can be improved using Map-Reduce framework. But it has been observed that the efforts are mostly done for the improvements in the extraction process while relatively low number of contributions are available on how efficiently this huge extracted data can be integrated.

As a result of completion of extraction process, a huge amount of super concepts and relationships are available in the form of Is-A pairs. Handling these billions of super concepts will also take considerable amount of time and resources. The most time consuming process is to find the similar Is-A pairs to be merged. There can be different cases of similarities between the pairs e.g.

1. two super concepts can exist with the same name and have significant similarity between their sub concepts.
2. two super concepts can exist with the same name but their sub concepts might have no similarity at all, which shows that both super concepts are representing different meaning and context.
3. two super concepts can exist with a slight difference in their names possibly due to a spelling mistake, but they have a significant similarity between their sub concepts.
4. two super concepts can exist with different names and have significant similarity between their sub concepts as well, which shows the possibility that both the super concepts are used in the same context and can be called as synonyms.

In short, during the integration process; time complexity should be the main focus as we want to make this process as efficient as the extraction process. Therefore, the best way to achieve efficiency is to formulate the similarity algorithms within Map-Reduce framework.

There are a variety of algorithms which can be used for finding similarities between datasets. A few important algorithms are discussed in the next chapter.

The main task of purpose thesis is to engineer an efficient process of performing set similarity joins with Map-Reduce framework.

Chapter 2

Foundations

This chapter highlights the foundations and concepts on which this thesis is based on. Knowledge Base, Taxonomy and Map-Reduce are discussed in detail in this chapter.

2.1 Knowledge Base

Knowledge base is a centralized repository of data which stores complex structured or unstructured information used by computer applications. Using knowledge bases for catering problems involving large data sets has become more common in the last decade as it provides a simple and easy way of information storage, management and retrieval. Large knowledge bases with high quality data sets can be used in many important tasks such as question answering and summarization.[1]

2.1.1 Knowledge Base Creation

Processing natural language text and generating relational datasets through information extraction techniques is a known way of creating knowledge bases. One way of creating a knowledge base involves two main processes. First is to extract the required information from the corpus and second is to integrate the extracted data in an efficient and organized way. Moreover, distributed data processing algorithms such as Map-Reduce through its parallel processing structure have made the extraction process more efficient. Therefore, the goal of creating a large knowledge base in an efficient way can be achieved by Map-Reduce.

[1] http://searchcrm.techtarget.com/definition/knowledge-base

2.2 Taxonomy

Taxonomy is a knowledge base used for classification of classes and objects in a hierarchical order and therefore considered to be the basis of classification schemes in information management. Taxonomies are generated for grouping things of same meaning, type or role etc. Tree or graph structures can be used for the representation of taxonomies where each node represents some concept or value. Taxonomies have become a very valuable resource in a wide range of applications involving data analysis and understanding and are widely used for information structuring for easier retrieval and management. A simple taxonomy [2] example in a tree structure is shown below.

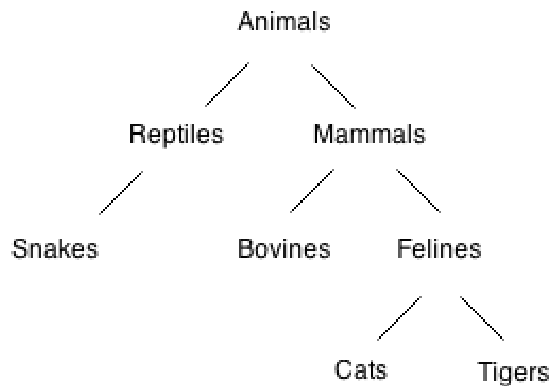

Figure 2.1: A Simple Taxonomy.

In the figure 2.1, 'Animals' is the root or super concept where as 'Reptiles' and 'Mammals' are its sub concepts or intermediate level concepts which are further divided into their sub concepts as well. The taxonomy grows further and as the new sub concepts are found, they are attached as subordinates of the relevant super concept.

[2] http://www0.cs.ucl.ac.uk/staff/a.hunter/tradepress/tax.html

2.3 Map-Reduce

The immense data available for processing in current era such as the crawled documents, web logs etc. greatly increases the requirement of distributed servers for data storage. There can be multiple computations that could be performed on this data to provide valuable information. These computations would be simple; however due to large amount of input data, the computations needs to be distributed across hundreds of machines so that the process can finish in a reasonable amount of time. The large number of computers in which huge dataset is distributed is collectively called as cluster.

To parallelize the computations on these large sets of data, a programming paradigm called Map-Reduce is used which helps achieve scalability across thousands of servers in the cluster. The framework was first introduced by Google in 2004 [5]. It was designed for computations over massive data sets. A programming model which enables development of large and scalable applications involving huge data processing in a parallelized way on large clusters and commodity hardware.

The Map-Reduce framework provides redundancy and fault tolerance. It also takes advantage of the locality of data by processing it on or near the storage assets so that the network cost can be reduced by avoiding/reducing the transmission distance of data. The frameworks hides the complexity of handling data and job partitioning from the end user which helps the user in concentrating on the main task rather than to focus on the complexities of distributed data processing like fault tolerance, load balancing, data distribution and task parallelization. [3]

2.3.1 Programming Model

Map Reduce interleaves both sequential and parallel computations in a specific functional style. It considers the basic unit of information as a key-value pair. The inputs to Map-Reduce Algorithm is provided as Key-Value pairs. The set of pairs are processed in three phases i.e. the map phase, the shuffle phase and the reduce phase.

Map stage consists of a map function which is defined by the user. The map function describes how the key and the value pairs can be obtained from the provided input data. This function is applied to each logical record of the input data set in order to compute a set of intermediate key-value pairs. Provided that every map operation is independent from others, therefore they can be parallelized. Although it is limited by the number of processing units and independent data sources.

The map operation must be stateless i.e, only one pair at a time should be processed which allows ease of parallelization as several inputs can be processed by different machines at the same time [8]. The signature of a map function are shown below.

$$map(k1, v1) \rightarrow list(k2, v2)$$

After map stage, the next is the shuffle stage in which the output Key-Value pairs of the map function are taken and all the intermediate values with the same intermediate keys are grouped together. This is done by a combiner function. The combiner can be a simple grouping function so that all values of a particular key can be processed together in the same machine. This combining occurs automatically and is seem less to the programmer. The signatures of a combiner function are shown below.

$$combine(list(k2, v2)) \rightarrow list(k2, list(v2))$$

The final stage consists of reduce function. Here the reducer takes the intermediate key and the set of value pairs as the input. The reduce function is described by the user in which he specifies how to obtain an output value from the input Key-(Value List) pair.

One of the sequential facade of Map-Reduce framework is that the reduce phase can only start once all the maps are completed. Since the combiner groups the keys, all the values which belong to a specific key are accessed by the reducer and sequential computations are performed on them. Moreover, the reducers operate on different keys at the same time which exploits parallelism.

$$reduce(k2, list(v2)) \rightarrow list(v3)$$

2.3.2 Execution Structure

Map reduce framework consists of a master node and many worker nodes. The input is splitted into multiple chunks of data. These input splits can be processed by different machines in a parallel way. A partition function such as hash(key) mode n, can be used for the splitting and the user can specify this function as well as the number of splits needed. The machines involved in Map-Reduce are called master and workers. The master machine takes the input, divides them into smaller sub-problems and distributes them to the worker machines. [4]

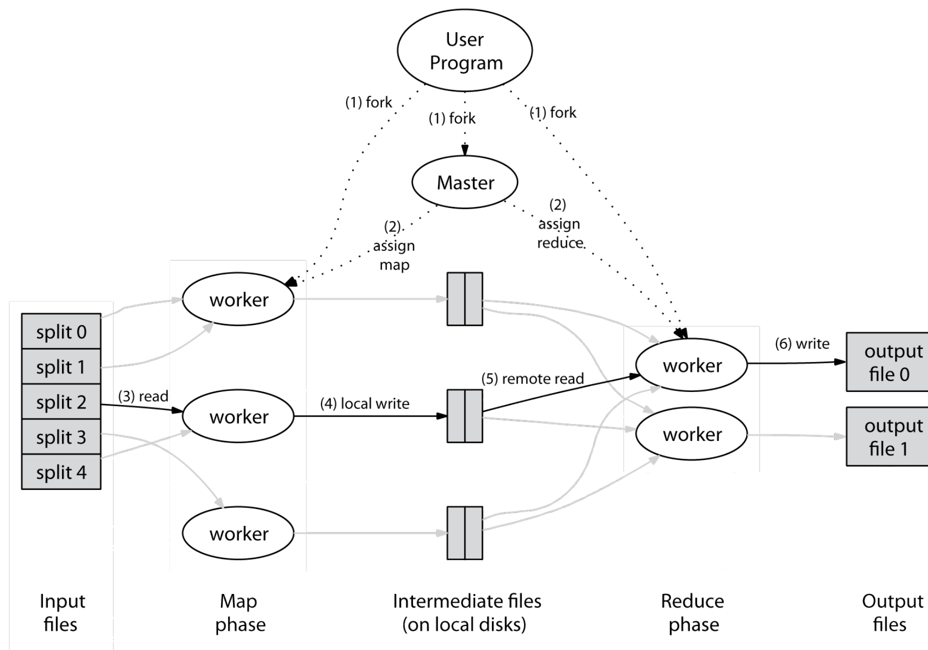

Figure 2.2: Execution Overview of Map-Reduce

After map task is assigned to workers, each worker node reads the splitted chunk of input data and passes each pair to user defined map function. The output of the map function is intermediate key-value pairs and does the desired map processing assigned to it. Workers write the output locally to its own disk. The output by the map workers form the intermediate files. The location of these files in the local disk is passed to the master so that the master can forward it to the reduce workers. The workers in the reduce tasks then read

the intermediate files and pass the data to the reduce function. When all map and reduce tasks are done, the master wakes up the user program. After completion of Map-Reduce, output will be available in output files equivalent to the number of reducers.

Chapter 3

Set Similarity Join Algorithms and Architecture

This chapter includes the description of the basic similarity finding algorithms and their comparison. The chapter also explains how these algorithms are integrated within the Map-Reduce framework to achieve efficient parallel processing of large datasets generated by the extraction process. Furthermore, the overall architecture of the system is also explained in this chapter.

3.1 Similarity Algorithms

As a result of the extraction process, we get huge data sets which can contain duplicate information and this data needs to be merged in such a way that all the unique information related to a specific concept should be found under the same single node in the knowledge base. But it's not the only case of finding similar data. As discussed in the problem statement, there can be different types of similarities between the extracted Is-A pairs.

A variety of algorithms are available for finding similarity between the data sets, we need to identify which algorithms can be efficiently integrated with Map-Reduce framework to achieve our goal of implementing 'Set Similarity Joins with Map-Reduce'. Therefore, we chose two of such similarity algorithms and discussed them in detail.

3.1.1 Jaccard Similarity Coefficient

Jaccard similarity coefficient, also known as Jaccard index is a statistical coefficient used to compare the similarity of finite sample sets. The algorithm was first introduced by Dr. Paul Jaccard in 1901. He called it *"coefficient de communauté"*. The formal definition [1] states:

"For two sets A and B, the Jaccard coefficient is equal to the length of intersection divided by the length of union of the sets A and B."

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

Figure 3.1: Jaccard coefficient for sets A and B

Whereas the intersection between two sets A and B gives all items which are common in both sets; and the Union between two sets A and B results in the items which are in either set.

For example, we have two sets A and B.

$A = \{0, 1, 3, 5, 7, 9\}$
$B = \{1, 4, 5, 6, 8, 9, 10, 14\}$

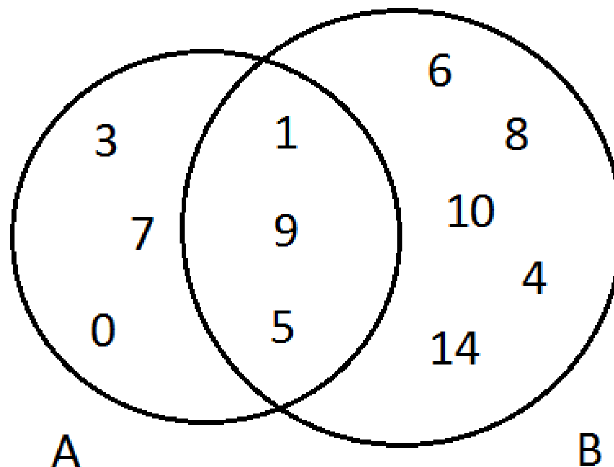

Figure 3.2: Sets A and B with Jaccard index 3/11

[1] $http://en.wikipedia.org/wiki/Jaccard_index$

The result of intersection between the elements of both the sets is 3 because 1, 5 and 9 are common in both sets while the result of union will be 11 which counts the unique entries of elements in both sets. So the Jaccard index will be computed as 3/11 as shown in figure 3.2.

In comparison to Jaccard index, **Jaccard distance** [2] is a complementary concept which is calculated by subtracting the Jaccard coefficient from 1 or by dividing the difference of the sizes of the union and intersections of the two sets by the size of the union. Figure 3.3 shows the mathematical formula below.

$$d_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}.$$

Figure 3.3: Jaccard distance for sets A and B

3.1.2 Dice Coefficient

Dice coefficient or Sørensen index is a statistical measure for finding the similarity or comparison between the two samples. It was developed by two botanists Lee Raymond Dice [7] and Thorvald Sørensen [12], published in 1945 and 1948 respectively. Mathematical formula for Sørensen index is shown in figure 3.4.

$$QS = \frac{2C}{A+B} = \frac{2|A \cap B|}{|A| + |B|}$$

Figure 3.4: Sørensen index for samples A and B

Where A and B are number of species in the samples and C is the number of species which are common in both samples. QS is The similarity quotient and it ranges from 0 to 1. The Formula can also be interpreted as a measure of similarity between two sample sets A and B.

[2] $http://en.wikipedia.org/wiki/Jaccard_index$

If we consider the example of sets A and B which we used for calculating the Jaccard index, the Dice coefficient will give us different results with the same sample sets A and B. As we calculated before, the result of intersection between the elements of sets A and B is 3. The length of sets A and B is 6 and 8 respectively. Putting these values in the formula in figure 3.4, we calculate Dice Coefficient as QS = (2 . 3) / (6 + 8) = 6 / 14 = 3 / 7.

String Similarity Measure

The algorithm can also be used as a string similarity measure, the coefficient can be calculated for two strings A and B using bi-grams as shown in the figure 3.5 [9] . Where n_t is the number of character bi-grams common in both strings and n_x and n_y are the number of bi-grams in the strings x and y.

$$s = \frac{2n_t}{n_x + n_y}$$

Figure 3.5: String Similarity Measure

Lets consider an example, to compute the similarity between 'plant' and 'plants'; and between 'plants' and 'player', the set of bi-grams for all the three strings are shown below;

$X = \{pl, la, an, nt\}$
$Y = \{pl, la, an, nt, ts\}$
$Z = \{pl, la, ay, ye, er\}$

For 'plant' and 'plants': Set X has 4 elements while Set Y has five elements. 4 bi-grams are common between the sets. Putting these values in the formula in figure 3.5, we calculate s = (2 . 4) / (4 + 5) = 8 / 9 = 0.89.

For 'player' and 'plants': Set Y and Z both have 5 elements. 2 bi-grams are common between the sets. Putting the values in the formula, we calculate s = (2 . 2) / (5 + 5) = 4 / 10 = 0.25.

The higher the result of Dice coefficient, the higher is the similarity between the strings.

3.1.3 Comparison between Jaccard and Dice coefficients

A lot of research has already been done related to the comparison of Jaccard, Dice and other coefficients. The reason we've only discussed Jaccard and Dice above is because we found both these algorithms recommended to be used instead of others. Both the algorithms are quite similar in the mathematical representation as well. It has been noticed that there is not a very significant amount of difference between their results as well when applied to the similar input of sets.

In [1], the authors did a detailed comparison on few similarity algorithms including Jaccard and Dice for finding genetic similarity between the pair of individual species. Both these algorithms are preferred to be used over the others, but there has been a slight difference between the results obtained by Jaccard and Dice coefficients. The values obtained from Dice are slightly higher than the ones obtained from Jaccard. It has been observed that both the coefficients shows different results from the others involved in the comparison. It can be seen by inspecting their mathematical representation that both have common principles which are different from the rest. The only difference between these two is the double weight that is given to the positive co-occurrences(intersection of sub items) in the Dice coefficient.

The bases on which the most appropriate coefficient should be chosen depends upon different criteria and can be domain specific as well. In our scope, we have to find similarity between the different sets on the basis of their sub items. Therefore, the criteria might change and we need to find the appropriate similarity algorithm as required by the process.

In the next section, the domain and type of comparisons/operations which are performed in this thesis and the selection of the appropriate coefficients are discussed.

3.2 Set Similarity Joins

We used set similarity joins for finding similar data sets and merging all of them under one super concept to avoid duplicate results. In this section, the basic algorithm used for set similarity joins is discussed in detail. How it is improved in order to achieve more efficient results as well as the most important part 'How this algorithm is integrated within the Map-Reduce framework' is also discussed. Furthermore, all the different types of similarities mentioned in the problem statement with the details of their working with Map-Reduce framework are also a part of this section.

3.2.1 Background Overview

Let's start from the step where we extracted the data from the web corpus in the form of sentences having keywords: "such as", "like", "including", "especially" etc. These keywords indicate that the sentence probably states something about any super concept and its sub concepts. For example;

$$plants\ such\ as\ tree, grass, flowers, tree\ are\ very\ common$$

$$plants\ like\ cactus, bamboo, holly\ are\ not\ very\ common$$

$$plants\ especially\ brewery, plastic, chemicals\ have\ heavy\ machinery$$

$$plant\ such\ as\ grass, sunflower, daisy, tree$$

Now we need to identify the super concept and its sub concepts from the sentences and organize them in the form of sets so that we can further process them for finding similarities between them. Once we process these sentences we'll get the data in the form of the data sets as shown below;

$plants = \{tree, grass, flowers, tree\}$
$plants = \{cactus, bamboo, holly\}$
$plants = \{brewery, plastic, chemicals\}$
$plant = \{grass, sunflower, daisy, tree\}$
$green = \{sunflower, daisy, tree, spinach, flowers, grass\}$

3.2.2 Similarity Cases

As mentioned in the problem statement, there are several cases that need to be catered while finding similarities between the sets. In this section, we discussed all the cases before hand that we are going to perform during the data integration process. At this stage the basic idea of the case and brief implementation details are presented. Later on, more details are presented on how they are working within Map-Reduce framework.

Duplicates Removal

During the data extraction phase, it might be possible that we find duplicate data values even within a set. In the data sets explained in the example in section 3.2.1. There exist a set in which an element ”tree” is found twice. Therefore, it's necessary to remove the duplicate values in order to build a knowledge base without duplicate sub concepts within a super concept. This is the first operation that is performed on individual data sets to ensure there are no duplicate values within a set.

Example: $plants = \{tree, grass, flowers, tree\}$

Sense Identification

While processing natural language you might come across a scenario where you find two similar words (same spellings) but their meanings are totally different because they are used in different contexts. In the same example of data sets in section 3.2.1, we also witnessed such a scenario where we found two occurrences of a super concept ”plants” but after having a look on their sets we came to realize that there is no similarity between them and both are used in different contexts and should not be merged. Therefore, the process of identifying the context or sense in which a super concept is used is also a very important aspect that is also taken care of during the process.

Example:

$plants = \{tree, grass, flowers, tree\}$
$plants = \{brewery, plastic, chemicals\}$

To identify the sense between the super concepts with similar names, we need to apply a basic similarity algorithm (Jaccard or Dice coefficient) between the data sets and if the value of the coefficient is 0 then there is a high possibility that their super concepts are representing different senses.

Spelling Difference Identification

A very common scenario in which you might find small spelling differences in some keywords within the data sets that will eventually create problems during the merging process. The difference can occur because of few reasons. It can be due to spelling mistakes or because the extracted data is from different web documents or can also be because of singular/plural versions of the keyword. The system will not allow the information to be merged until or unless it's a perfect match between the two keywords. Therefore, it's also one of the very important cases that are handled in the processing.

This case can be handled in a later stage during the merging process. At first we prefer the processing of exact matches and then we can process this case on the few remaining non-merged data sets.

Example:

$plants = \{tree, grass, flowers, tree\}$
$plant = \{grass, sunflower, daisy, tree\}$

To solve such a case, our approach is to apply a string similarity algorithm (as described in section 3.1.2: String Similarity Measure) on the super concepts and if the threshold of similarity is fulfilled then there is a chance that this can be a spelling difference case. The next step is to apply the set similarity algorithm and if it fulfills the required threshold then its means the super concepts are same and should be merged.

Synonym Identification

There can also be a possibility for two data sets that their super concepts don't match with each other but there is a significant amount of similarity in their sub concepts. In such a case, it can be said that both the super concepts are representing same senses as there is a significant overlap in their sub concepts. Therefore, both can be referred as synonyms of each other and the solution is to either merge them or at least mention the name of the other synonym in case if data is not merged.

Example:

$plant = \{grass, sunflower, daisy, tree\}$
$green = \{sunflower, daisy, tree, spinach, flowers, grass\}$

The solution for this case is simple and should also be handled in the later stage. The approach is to compare the sub concepts of both super concepts and if there is a significant amount of similarity then both should be declared as synonyms of each other.

3.2.3 Basic Set Similarity Algorithm

The basic algorithm that is used to identify the similarity between the data sets is Jaccard coefficient and in the later stage when the initial iterations of set similarity are completed, Dice coefficient will be used for finding the similarity between the strings (name) of the super concepts.

We need to set a threshold value for Jaccard coefficient which will be considered to be a measure of realistic similarity between the data sets. But we cannot do it simply as the size difference between the sets also plays an important part in the calculation of Jaccard coefficient. Let's consider some examples of set similarity joins between a couple of sets.

CASE 1:
$plants = \{tree, grass, cactus, bamboo\}$
$plants = \{cactus, bamboo, holly, grass\}$

The length of the sets is exactly same and there is no duplicate entry in any set. Jaccard coefficient will be calculated as: J = 3 / 5 = 0.6.

CASE 2:
$plants = \{tree, grass, rye, tree, tea, ferns, keek, neem, osage, holly, cactus\}$
$plants = \{cactus, bamboo, holly, grass\}$

In the first data set, there is a duplicate entry 'tree' which should be removed as per the similarity case 'Duplicates Removal'. The length of the sets is different with each other. One has 10 while other has 5 elements. Jaccard coefficient will be calculated as: J = 3 / 11 = 0.27.

If you notice, the second data set in both the cases is same and the length of intersection of the sets is also same i.e. 3 in both cases. But the results are quite different. This is because there was a significant difference of sizes between the sets in case 2. Therefore, Jaccard coefficient has a relatively lower value in this case.

Issues

Since we cannot guarantee about the size difference between the data sets and we still need to set a threshold value which specifies the measure if there is enough similarity between the sets to be merged. Therefore, we need to identify all those factors on which the result of Jaccard coefficient depends.

Let's consider the example in case 2 again with a different perspective. Assume that the threshold for similarity measure is 0.5. The length of intersection is 3 and the size of the smaller set is 4 which means 3/4 th part of this set is similar with the bigger set and realistically it should be merged but the value of Jaccard coefficient is 0.27, which is less than the threshold and merge will not be allowed. Hence, if there is a significant amount of difference of sizes between the sets then the percentage of similarity of the smaller set with the bigger set should also be counted in defining the threshold value for similarity acceptance.

Improvements

The possible issues in the basic set similarity algorithm are identified and a conclusion is made that the value of Jaccard coefficient as a threshold of similarity should not be fixed. As the difference of sizes between the sets is changed abruptly and we cannot guarantee realistic results in the process of merging the sets. Therefore, the threshold will be decided on runtime depending on the criteria below.

1. The size difference between the sets.
2. The percentage of the number of elements in smaller set which are similar with the elements of larger set (in case of significant difference between the sizes of the sets).

3.3 Set Similarity Joins with Map-Reduce

The complete set similarity join process consists of 4 different algorithms based on the cases specified in the problem statement. The overall process is divided into three sub processes. To reduce the complexity, each process will have individual Map-Reduce jobs designated to perform a specific similarity case. The processes are listed below.

1. Set Similarity Identification.
2. Spelling Difference Identification.
3. Synonym Identification.

3.3.1 Set Similarity Identification

This is the first step of the data integration phase and also the main process in performing the set similarity joins with Map-Reduce. It is responsible for identifying the similarity between the data sets and merging them. The process covers the following cases.

- Duplicates Identification
- Sense Identification
- Similarity Identification

Let's consider a small example. The input file contains the following data sets to be processed.

$fruits >> A, B, C, D$
$factory >> X, Y, Z$
$fruits >> B, D, E$
$factory >> W, Y, X$
$fruits >> C, A, G$

Working of Map Function

The Map function iterates over all the rows of the input file one by one. It works on a simple algorithm of extracting the super concept and its sub concepts from each row it reads. Then it writes the 'super concept' as key and the complete string containing the 'sub concepts' as value.

In the example of input data in section 3.3.1, there are two super concepts i.e. 'fruits' and 'factory' having 3 and 2 rows each respectively. Figure 3.6 shows the overall working of this process.

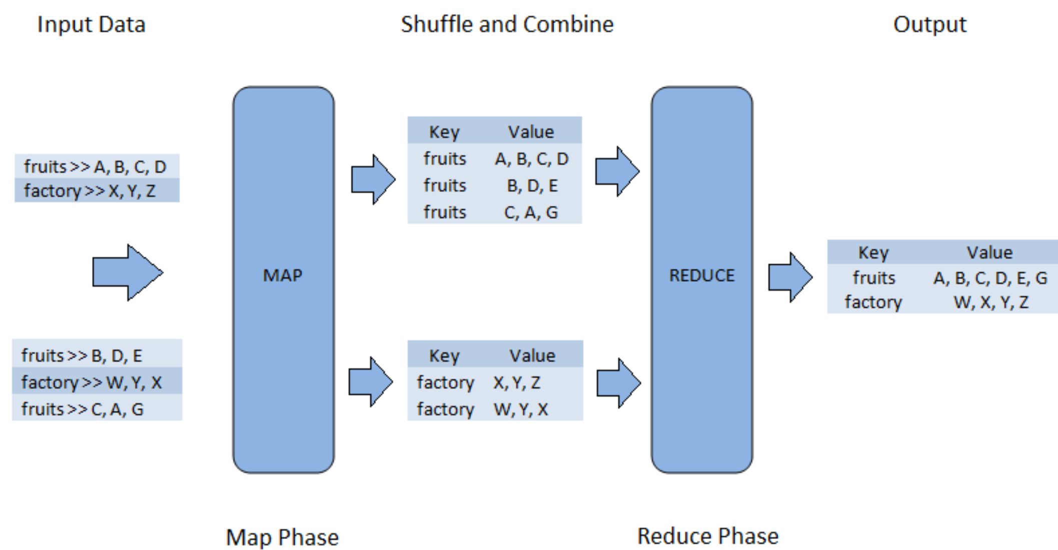

Figure 3.6: Execution Overview of Map-Reduce

Working of Reduce Function

The number of reduce tasks depends upon the number of keys mapped by the map function. In this case there will be two reduce tasks, one for key 'fruits' and the other is for key 'factory'. The Reduce task will get the key along with a list of all the values for this key mapped by the map function. Each reduce task will perform the following steps.

```
create a list 'LIST' of hash sets;
while input has more tuples do
    read current tuple;
    extract super and sub concepts;
    create a hash set 'A' of sub concepts;
    while LIST has hash sets do
        read the hash set;
        if intersection with set A then
            calculate Jaccard index;
            calculate size difference;
            calculate percentage similarity;
            if calculated values equals/exceeds threshold values then
                merge the sets;
            else
                add set A in the list;
            end
        end
    end
end
```

Algorithm 1: Algorithm for Reducer in Set Similarity Identification

Since, every super concept is mapped as a key and each key has a separate Reduce thread. Consider an example of two super concepts 'plants' and 'plant' with a significant similarity in their sub concepts. This process will not be able to compare their sets for similarity identification as both keys are considered different (due to a slight difference in spellings) and will have separate Reduce threads.

3.3.2 Spelling Difference Identification

At this stage, the major process for merging similar data sets has been completed by set similarity identification process. The next phase is to identify all those data sets which were left and not included in the process because of small difference in spellings in their super concepts.

The approach is to apply the Dice coefficient to find similarity between the super concepts. In this process we also need to fix a threshold value of Dice coefficient which will be considered as a measure of significant similarity between the two super concepts. Consider the example shown below.

$fruits >> A, B, C, D, E, F, G$
$fruit >> H, E, I, A, B$

In simple words, the algorithm for this process is to identify string similarity between the super concepts and if significant similarity exists then we need to compare their data sets to find similarity between them as well. If the data sets are also similar then they will be merged. Implementation of this algorithm within Map-Reduce framework is complex. Therefore, to reduce this complexity, the process is divided into two Map-Reduce jobs. The execution overview of the first Map-Reduce job is shown in figure 3.7.

First Job: Working of Map Function

The map function performs the following steps.

```
while input has more tuples do
    read current tuple;
    extract super concept;
    create bi-grams of super concept;
    foreach bi-gram do
        write bi-gram as key;
        write complete input string as value;
    end
end
```

Algorithm 2: Map Function in Spelling Difference Identification

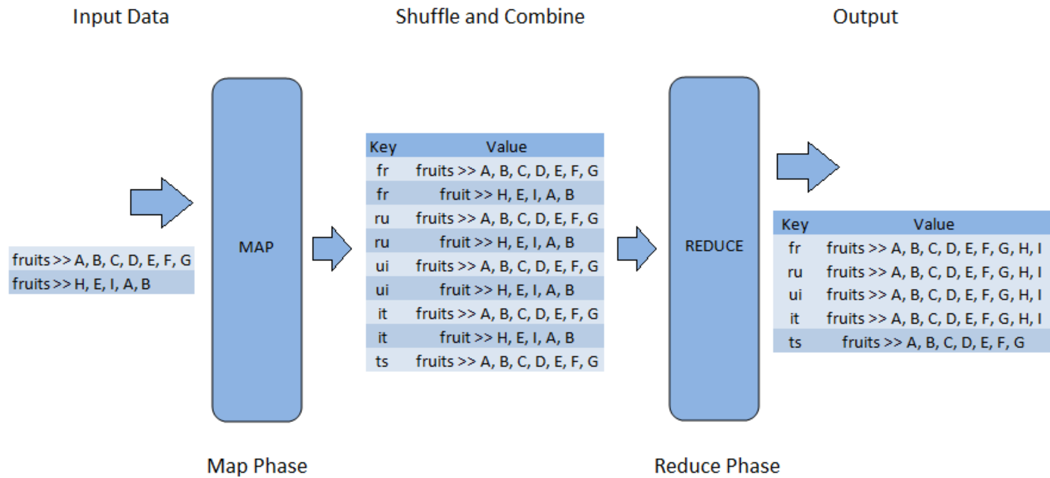

Figure 3.7: First Map-Reduce Job for Spelling Difference Identification

Consider the example in section 3.3.2, The Map function will read first line and extract the super concept 'fruits' and create its bi-grams as $\{fr, ru, ui, it, ts\}$. For each bi-gram as key it writes the complete input string as value. It does the same for the remaining input lines as well. Then shuffling of records takes place and results of output of map phase can be seen in figure 3.7. The purpose of writing complete input line as 'value' is to keep the record that the key belongs to a specific super concept and its data set.

First Job: Working of Reduce Function

The reduce function performs the following steps.

while *input has more tuples* **do**
    read current tuple;
    extract sub concepts and store in a hast set;
end
foreach *hash set* **do**
    compare every hash set with others;
    apply 'Set Similarity Identification' algorithm;
    merge the sets if significant similarity found;
end

Algorithm 3: Reduce Function in Spelling Difference Identification

The reduce function will extract and create a hash set of all 'sub concepts' one by one, apply set similarity algorithm and merge the sets. The same computations are performed in other reducer with keys 'ru', 'ui' and 'it'. But not for 'ts'; because there is only one value in the reducer for 'ts'. The value remains the same and no operations are performed within the reducer as it's the only value. The output of the process can be seen in figure 3.7.

At the completion of First Map-Reduce job, the data sets with small spelling differences in their super concepts have been merged but there are duplicate results in the output which needs to be removed.

Second Job: Working of Map Function

The output of the first Map-Reduce job with becomes the input for the second job. The Map function for the second Map-Reduce job performs the following steps.

```
while input has more tuples do
    read current tuple;
    extract bi-gram and its value;
    foreach bi-gram do
        write bi-gram's value as key; write bi-gram as value;
    end
end
```

Algorithm 4: Map Function in Spelling Difference Identification

The map function will extract the bi-gram and its value from the input line and then write the value as key and bi-gram as value to the output. It does the same with the remaining inputs. Then the process of shuffling of records takes place and the output of the map function can be seen in figure 3.8.

It can be observed that there are two different keys in the output . Therefore, there will be two reducer threads. One will get 'fr', 'ru', 'ui' and 'it' in the list of values while the other will only have one value 'ts'.

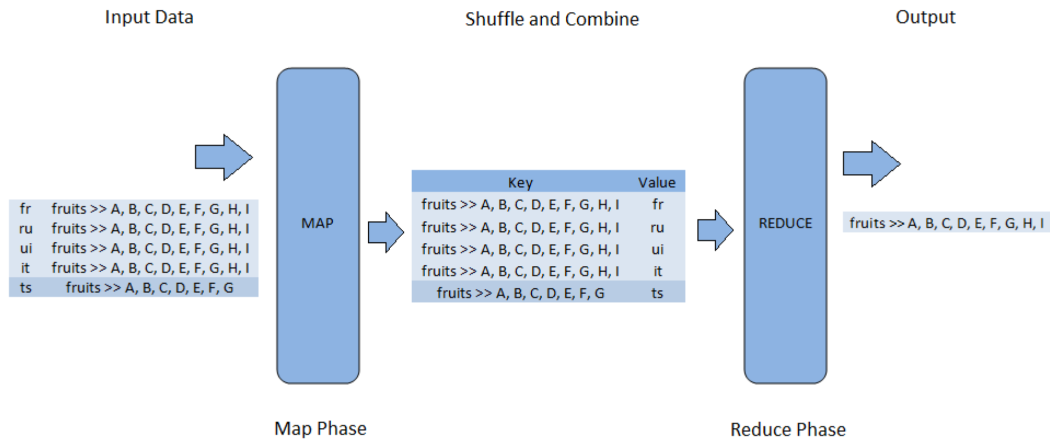

Figure 3.8: Second Map-Reduce Job for Spelling Difference Identification

Second Job: Working of Reduce Function

The Reduce function performs the following steps.

while *input has more tuples* **do**
  read current tuple;
  extract bi-gram from list of values and store in a hash set 'A';
  extract the super concept;
  create bi-grams of super concept and store in a hash set 'B';
  apply 'Set Similarity Identification' algorithm between sets A and B;
  if *significant similarity* **then**
    write the key to output;
  end
end

Algorithm 5: Reduce Function in Spelling Difference Identification

Table 3.1 shows the detailed step wise execution of both reducers. In the first reducer, $\{fr, ru, ui, it, ts\}$ and $\{fr, ru, ui, it\}$ are compared while in the second reducer $\{fr, ru, ui, it\}$ and $\{ts\}$ are compared for similarity. The first reducer will write the key as output because the hash sets are similar while the second reducer will output nothing.

The final output of the second Map-Reduce can be seen in figure 3.8. The output shows that even with the spelling difference in the super concepts (example in section 3.3.2), the algorithm identified the similarity between the sub concepts and merged the sets while neglecting the small difference of spellings.

Reducer No. 1	
Step 1	{fr, ru, ui, it}
Step 2	fruits
Step 3	{fr, ru, ui, it, ts}
Step 4	Similarity Found, Output: Key

Reducer No. 2	
Step 1	{ts}
Step 2	fruit
Step 3	{fr, ru, ui, it}
Step 4	No Similarity Found

Table 3.1: Step wise execution of Reduce Function

3.3.3 Synonym Identification

In this process, all the data sets will be compared with each other to find if similar data sets exists under different super concepts. The process covers the similarity case no. 4 mentioned in the problem statement. The approach is to compare the sub concepts and if there is significant similarity between them then their super concepts will be marked as synonyms of each other. Consider the following example which will be discussed throughout this section.

$fruits >> A, B, C, D, E, F$
$fresh >> F, E, G, A, B$

In this process, sub concepts are extracted from the input and each sub concept is written to the output as key while complete input string as value. Implementation of this algorithm within Map-Reduce framework is complex. Therefore, to reduce this complexity, the process is divided into two Map-Reduce jobs.

First Job: Working of Map Function

The Map function performs the following steps.

```
while input has more tuples do
    read current tuple;
    extract sub concepts and store in a list;
    foreach sub concepts in list do
        write sub concept as key;
        write complete input string as value;
    end
end
```

Algorithm 6: Map Function in Synonym Identification

Consider the example in section 3.3.3, The map function will read first line and extract the sub concepts $\{A, B, C, D, E, F\}$. Then for each sub concept, it writes the complete input string as value and sub concept as key to the output. The remaining inputs are also treated the same way and then shuffling of records is done. The output of the map function can be seen in figure 3.9.

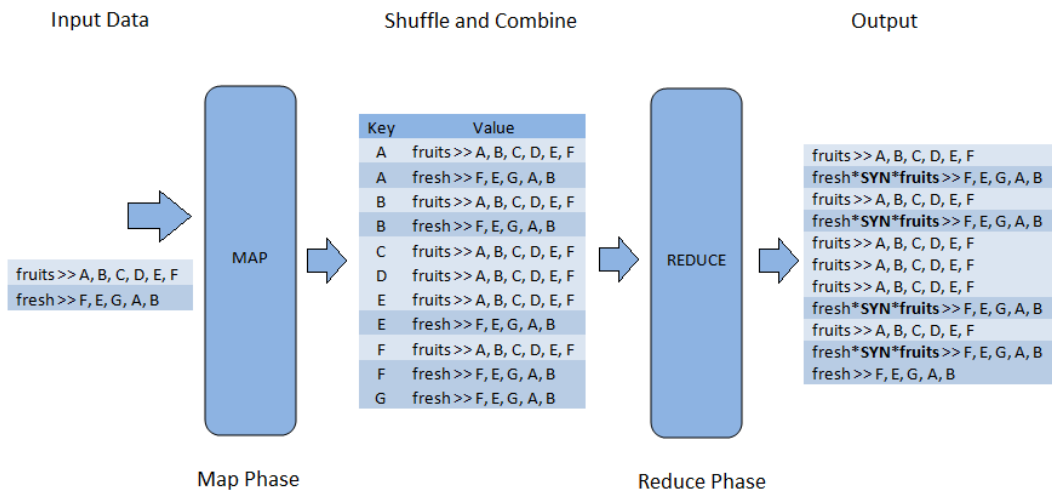

Figure 3.9: First Map-Reduce Job for Synonym Identification

First Job: Working of Reduce Function

The Reduce function performs the following steps.

```
while input has more tuples do
    read current tuple;
    extract sub concepts and store in a hast set;
end
foreach hash set do
    compare every hash set with others;
    apply 'Set Similarity Identification' algorithm;
    if significant similarity then
        modify super concept (text);
        write value to output;
    else
        write value to output;
    end
end
```

Algorithm 7: Reduce Function in Synonym Identification

The reduce function will extract the sub concepts and apply the 'Set Similarity Identification' algorithm. If significant similarity is found, the super concept will be modified. The modification is done for the identification that the super concept 'fresh' is a synonym of 'fruits'.

It can be observed that there are few records from the output of map function which have only one key/value pair and therefore their respective reducer tasks will also have only one value which cannot be compared with others. Records with key 'C', 'D' and 'G' are single key/value pairs. In this case their values are simply written to the output. Same computations will be performed in other reducers as well. The overall result of the first Map-Reduce job can be seen in figure 3.9.

At the completion of First Map-Reduce job, the data sets with possibility of synonyms in their super concepts have been identified but there are duplicate results in the output which needs to be removed.

Second Job: Working of Map Function

The Map function performs the following steps.

while *input has more tuples* **do**
    read current tuple;
    extract super concept;
    extract string containing sub concepts;
    foreach *super concept* **do**
        write sub concepts' string as key;
        write super concept as value;
    end
end

Algorithm 8: Map Function in Synonym Identification

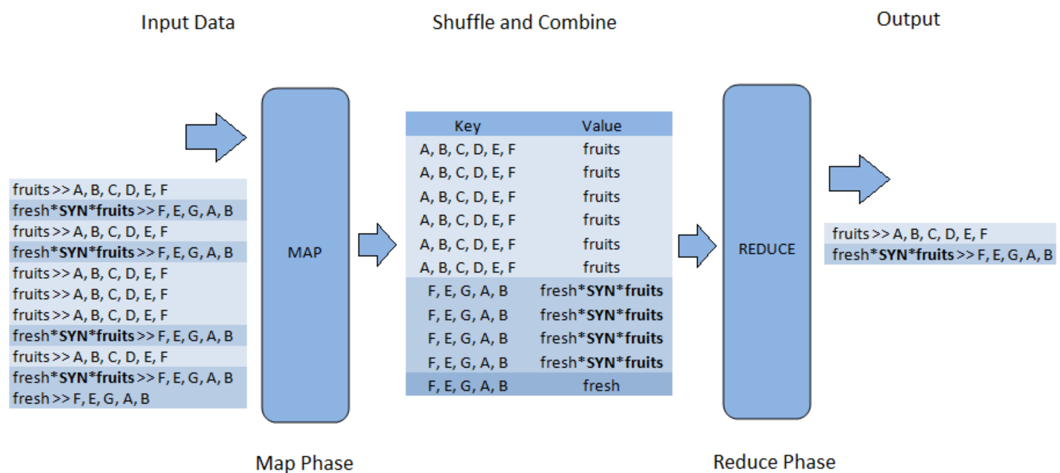

Figure 3.10: Second Map-Reduce Job for Synonym Identification

The map function will extract the super and sub concepts from the input and then write them to the output as value and key respectively. Consider the output of first Map Reduce job. The key/value pairs which the function will write to output are shown in figure 3.10.

Second Job: Working of Reduce Function

The list of values in each reducer contains the super concepts. We need to find all the unique super concepts. Therefore, we are using hash sets which only store a value if it's not in the set already. But as we are modifying some super concepts in the previous Map job, there is still a possibility that we get duplicates as well.

For example, in one of the reducers the list of values contain 'fresh*SYN*fruits' and 'fresh'. The hash set will add both the values because of difference in text but in reality both the values are representing a single super concept. Therefore we need to identify if the values are actually unique. The reduce function performs the following steps.

```
while input has more tuples do
    read current tuple;
    store the value (super concept) in a list;
end
foreach value in list do
    if value contains '*SYN*' then
        extract super concept without synonym;
        compare it with all other values (super concepts);
        apply 'Spelling Difference Identification' algorithm;
        if significant similarity then
            keep the super concept with '*SYN*';
            delete all entries of the same super concept without synonym
            modification;
        end
    end
end
```

Algorithm 9: Map Function in Synonym Identification

The algorithm is explained by using an example of "*SYN*" just to differentiate between the super concepts with synonym modification and without modification. It can be changed without any major modifications in the algorithm.

Reducer No. 1	
Key	**Values**
F, E, G, A, B	fresh*SYN*fruits
	fresh

Reducer No. 2	
Key	**Value**
A, B, C, D, E, F	fruits

Table 3.2: Reducer

Table 3.2 explains the execution of the reducer tasks in more detail for better understanding. The first reducer in has two values. As per step 2, The value is checked if it contains "*SYN*". Then the super concept will be extracted and compared with the other value. As they are similar, the value "fresh*SYN*fruits" will be kept while the other value will be removed from the set.

The second reducer in table 3.2 has only one value. Therefore, no comparison operations will be done here. The overall output of the second Map-Reduce job is shown below.

$fresh * SYN * fruits >> F, E, G, A, B$
$fruits >> A, B, C, D, E, F$

The overall integration process completes at the end of this phase. The extracted data-sets are reduced by the merging during the set similarity joins phase. The resultant data-set contains unique super and sub concepts.

3.4 Architectural Overview

The overall architecture of the system is shown in the figure 3.11. There are two main processes in the system i.e. Data Extraction and Data Integration.

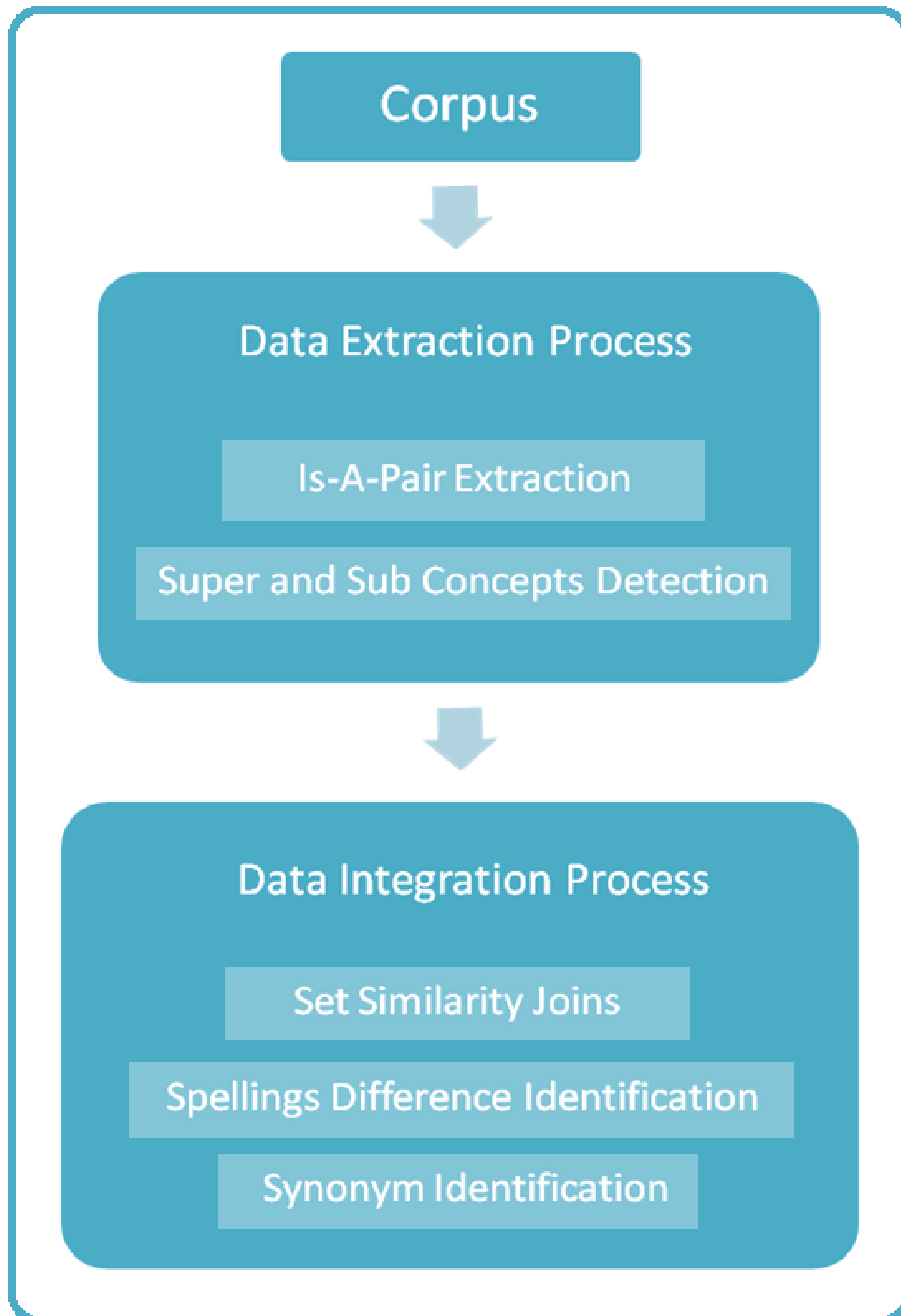

Figure 3.11: Architectural Overview

Chapter 4

Evaluation and Analysis

This chapter includes the overall experimental evaluation of set similarity join algorithms. It also concludes the optimal threshold values for Jaccard and Dice coefficient used in the experiments.

4.1 Threshold for Set Similarity Joins

As discussed in section 3.2.3, Jaccard index is not sufficient enough for defining an optimal threshold for finding similarity between the sets. Because in our case, it is fairly possible that the sets might differ in size and the results of Jaccard index varies with the variation of size difference of the sets under comparison. Therefore, the process of finding an optimal threshold for set similarity joins not only depends on Jaccard index value but also on the size difference between the sets. In case of significant difference between the sizes, another factor needs to be considered in defining the threshold which computes the percentage of the number of elements in smaller set which are similar with the elements of larger set. Term 'Percentage Similarity' is used for this factor.

Several experiments are performed with variations in the size difference of the sets in order to find appropriate threshold values. Table 4.1 shows some of the experiments depicting the change of Jaccard index with the variation of set size. In the table, column 'Similarity' represents the term percentage similarity discussed above.

In the experiments shown below, three different pair of sets are used for comparison having various size differences between them. Jaccard index (JI) is calculated thrice for every pair with a slight increase of similar elements between the sets. As the number of similar elements is increased, the value for JI is also increased. But on the other hand, JI values are decreased as the size difference between the sets is increased. In experiments 1(i), 2(i) and 3(i); it can be observed that the size difference is increased respectively but the value for JI is decreased.

Experiments for Threshold Analysis				
No.	**Elements of Set**	**Jaccard**	**Size**	**Similarity**
1(i)	**A**, B, C, D, E, F, G	0.076	1	15%
	A, H, I, J, K, L, M			
1(ii)	**A**, **B**, C, D, E, F, G	0.167	1	33%
	A, **B**, X, I, J, K, L			
1(iii)	**A**, **B**, **C**, D, E, F, G	0.27	1	43%
	A, **B**, **C**, I, J, K, L			
2(i)	M, N, **O**, P, Q, R, S, T, U, V, W	0.067	>2	20%
	O, X, Y, Z, A			
2(ii)	M, N, **O**, **P**, Q, R, S, T, U, V, W	0.142	>2	40%
	O, **P**, X, Y, Z			
2(iii)	M, N, **O**, **P**, Q, **R**, S, T, U, V, W	0.23	>2	60%
	O, **P**, **R**, Y, Z			
3(i)	**D**, E, F, G, H, I, J, K, L, M, N, O, P, Q	0.058	>3	25%
	A, B, C, **D**			
3(ii)	**D**, **E**, F, G, H, I, J, K, L, M, N, O, P, Q	0.125	>3	50%
	A, B, **E**, **D**			
3(iii)	**D**, **E**, **F**, G, H, I, J, K, L, M, N, O, P, Q	0.2	>3	75%
	A, **D**, **E**, **F**			

Table 4.1: Experiments for Threshold Values

An overall decrease in the Jacard index values is seen in experiments 1 to 3 because the size difference of set is increased. Therefore, considering all these factors, mean values are calculated for the Jaccard Index and percentage similarity from all the experiments and a threshold for similarity is defined as shown in table 4.2.

Optimal Threshold Values for Set Similarity		
Size Difference	**Jaccard Coefficient**	**Percentage Similarity**
1 - 2 Times	>0.15	>30%
2 - 3 Times	>0.1	>50%
3 - 4 Times	>0.08	>60%
>4 Times	>0.05	>65%

Table 4.2: Threshold Values for Set Similarity Identification

4.2 Threshold for String Similarity

Several experiments have been done in finding an appropriate threshold value for Dice coefficient to be used for finding similarity between two strings. Table 4.3 shows a sample of experiments of a string with several similar strings.

Threshold Analysis of Dice Coeffient			
No.	**String A**	**String B**	**Dice Coefficient**
1	plant	plants	0.88
2	plant	planted	0.8
3	plant	slant	0.75
4	plant	chant	0.5
5	plant	plan	0.85

Table 4.3: Threshold Analysis for String Similarity

There is always a possibility that the selected minimum threshold value declares several pairs of strings as similar but in reality they are not. Since the algorithm is used here for identifying the possibility of spelling difference/mistake between two super concepts. Even if the computation declares them as similar, the process will not allow them to be merged until there is a significant

similarity between their sub concepts. Therefore, as per experiments, a minimum threshold value is set to be at least **0.7** for two strings to be considered as similar.

4.3 Environment Details

Experiments are conducted on the system with specifications details shown in figure 4.1.

Type	Details
CPU	AMD E-350 Processor
Clock Speed	1600 MHz
Cores	2
Memory	4 GB
Cache	512 KB
Linux Kernel	Ubuntu 11.10 (GNU/Linux 3.0.0-12)

Figure 4.1: System Specification Details

All machines are running Apache Hadoop 1.0.3. File system used in all experiments is HDFS.

Type	Details
Data Nodes	15
Name Nodes	1
Job Tracker	1

Figure 4.2: Cluster Details

4.4 Set Similarity Joins

Several experiments are performed for measuring the efficiency of set similarity join process. In each experiment, a Map-Reduce job is executed with an increase in the number of input records relatively larger than the previous job. Figure 4.3 shows detailed results of experiments performed. The first column shows the approximate size (in Mega bytes) of the input files used in each job. Input and output records with number of mappers and reducers for each job are also listed. It can be seen that the number of mappers and reducers are increasing with the increase in the input records.

Details of Experiments

Input Size (MB)	Input Records	Elapsed Time(s)	Output Records	Mappers	Reducers
≈ 2000	142688	51	2220	2	2
≈ 4000	299638	51	4519	4	3
≈ 6000	445584	51	6467	6	4
≈ 8000	587346	54	8430	8	5
≈ 10000	722591	54	10249	10	6
≈ 12000	864987	54	11909	12	7
≈ 14000	1017295	54	13943	14	8
≈ 16000	1162800	58	15753	16	9
≈ 18000	1313613	61	17407	18	10
≈ 20000	1458129	65	19175	20	11
≈ 22000	1595670	65	20688	22	12
≈ 24000	1750601	65	22333	24	13
≈ 26000	1895291	69	24029	26	14
≈ 28000	2040586	72	25605	28	15
≈ 30000	2191391	73	27337	30	16
≈ 32000	2337401	77	28978	32	17
≈ 34000	2482411	81	30686	34	18
≈ 36000	2631041	82	32329	36	19
≈ 38000	2774589	83	33810	38	20
≈ 40000	2888350	84	34904	40	21
≈ 42000	3031938	88	36434	42	22
≈ 44000	3171830	92	37955	44	23
≈ 46000	3303678	95	39349	46	24
≈ 48000	3455075	99	40947	48	25
≈ 50000	3596664	103	42493	50	26

Figure 4.3: Overview of Experiments for Set Similarity Joins

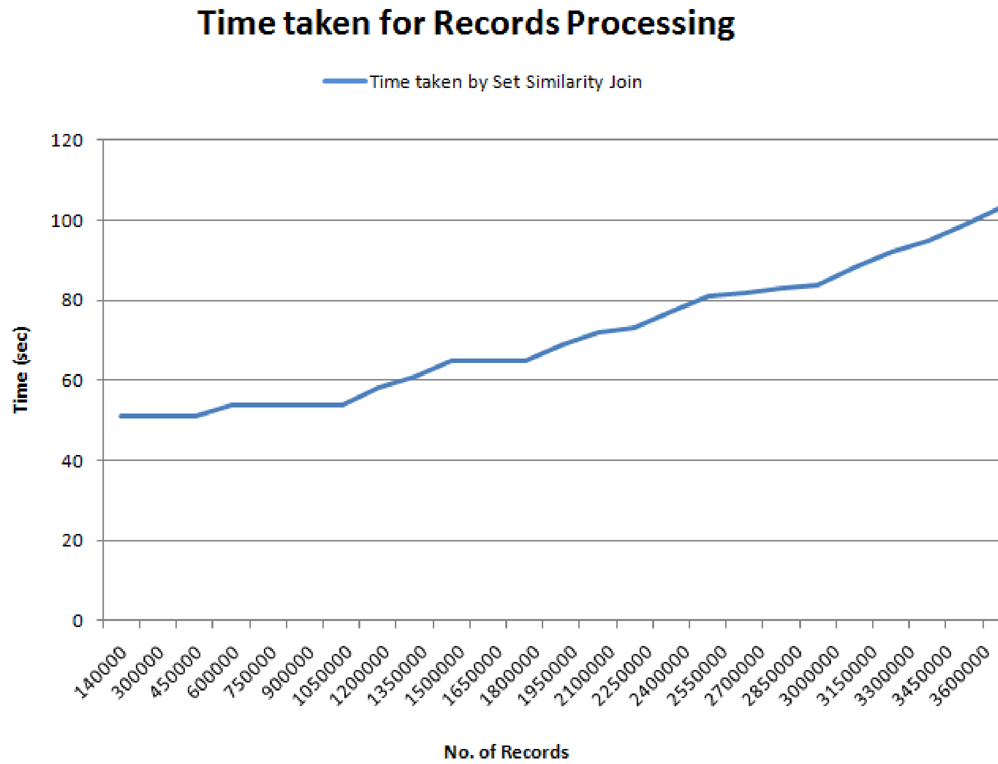

Figure 4.4: No. of Records processed with respect to time

Figure 4.4 shows the graph plot from these experiments for time taken by the Map-Reduce jobs with respect to number of input records. A significant amount of input records are increased during the iterations. The first job is executed with an input of approx. 140000 records and the process is completed in 51 seconds. As it can be seen in the graph that input records are significantly increased in the proceeding experiments but time taken for completion is not increased with the same ratio. In the last experiment, approx. 3600000 records are processed in only 103 seconds which is relatively very less time as compared to the time taken by the initial experiments.

It shows that the efficiency of the process is increased with the increase in input records. As mentioned before, the number of map and reduce tasks are also increased with the increase in input records which depicts the scalability of the whole process.

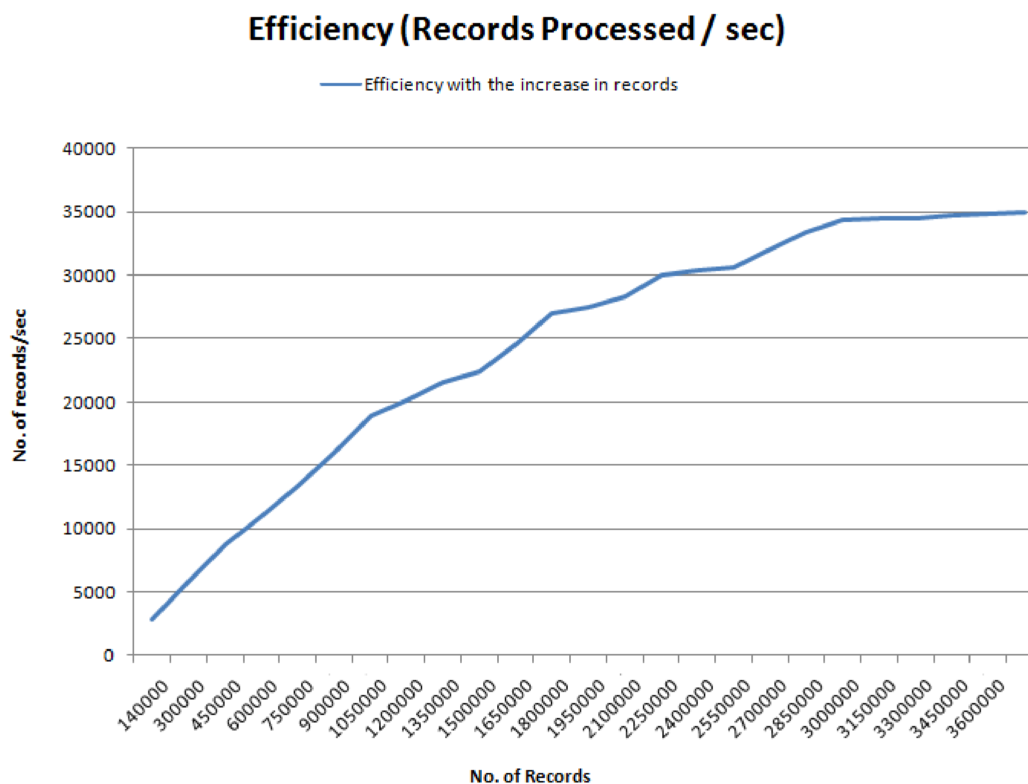

Figure 4.5: Efficiency Overview

The efficiency graph is shown in figure 4.5. It can be observed that number of records processed per second is increased with the increase in the input size until a point where approx 34000 records are processed per second. After this point, no further improvements are seen. This is because the number of map and reduce tasks exceeds the overall execution capacity of the cluster. Therefore, the exceeding tasks have to wait until there is a free place on the cluster for a new task to be executed.

Few other experiments are performed with duplicate input records to check the output of the process. Figure 4.6 shows a table with input records processed by different Map-Reduce jobs. The result of each job is shown in the output column. Similar input files are used to increase the overall count of the input records but the process identified the duplicate records and the output was same for every job. This shows that the process not only efficiently finds the similarities between the sub concepts but also ignores repetition of similar super concepts.

No. of Input Records	Time taken(s)	No. of Output Records
587346	55	8429
729108	55	8429
1316454	64	8424
1903800	67	8429

Figure 4.6: Experiments with Duplicate Input Records

4.5 Spelling Difference Identification

Several experiments are performed for measuring the efficiency of spelling identification process. Figure 4.7 shows detailed results of experiments. Elapsed time, input and output records are listed. Number of mappers and reducers are increasing with the increase in the input records.

Details of Experiments

Input Records	Output Records	Elapsed Time (s)	No. of Maps	No. of Reducers
2220	2098	128	2	2
4519	4200	182	3	3
6467	6004	241	4	4
8430	7758	285	5	5
10249	9364	340	6	6
11909	10874	376	7	7
13943	12689	433	8	8
15753	14279	468	9	9
17407	15759	498	10	10
19175	17364	543	11	11
20688	18705	571	12	12
22333	20186	612	13	13
24029	21688	650	14	14

Figure 4.7: Overview of Experiments for Spelling Difference Identification

The efficiency graph for this process is shown in figure 4.8. The number of records processed per second is increased with the increase in the input size. It can be noticed from the graph that the percentage of increase in efficiency is decreasing in the experiments performed with higher number of records which shows that at some point the efficiency will become stable when the limit of the cluster will reach.

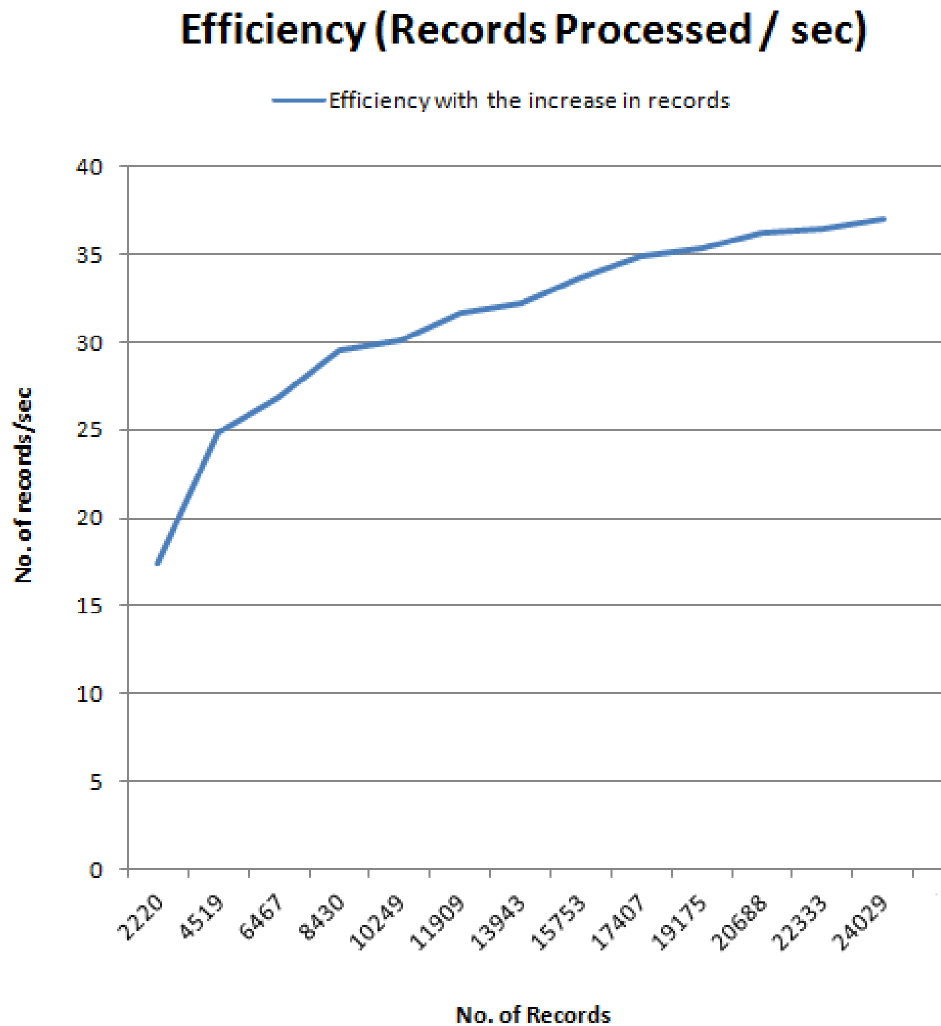

Figure 4.8: Efficiency Overview

The overall number of records reduced in this process is shown in a graph in figure 4.9. The total numbers of records are decreased after being processed which shows that minor spelling mistakes existed in the records as well.

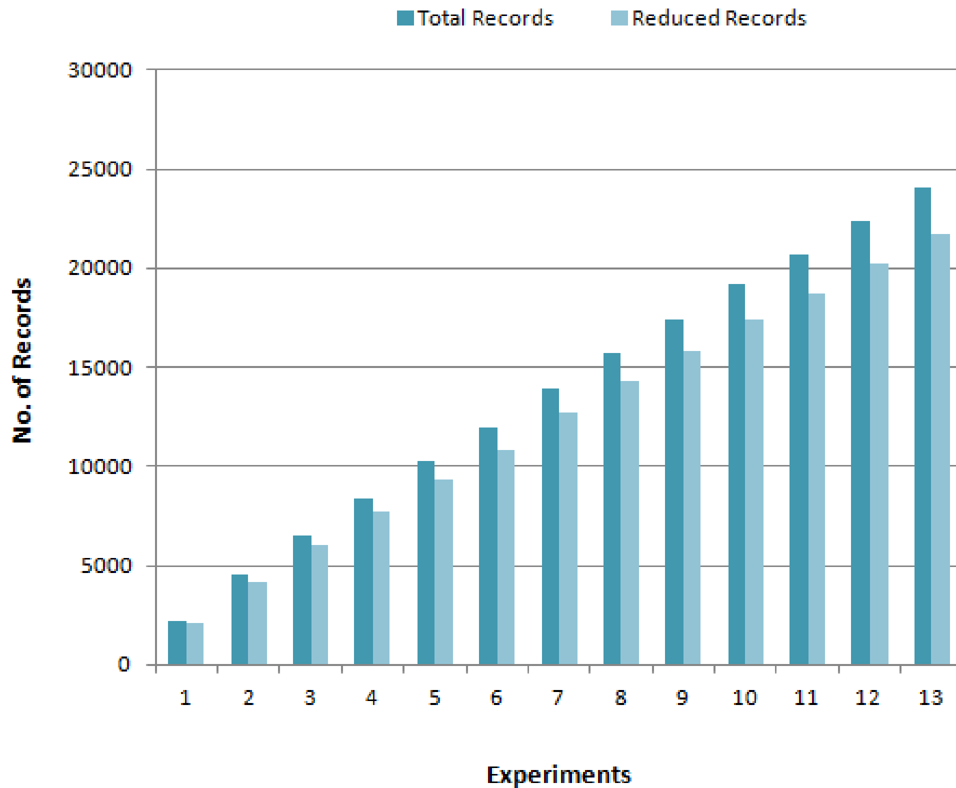

Figure 4.9: Overview of Records Reduced

4.6 Synonym Identification

The details of the experiments performed for synonym identification process are shown in the figure 4.10. Elapsed time, input and output records are listed. Number of mappers and reducers are increasing with the increase in the input records.

The graph depicting efficiency for this process is shown in figure 4.11. The number of records processed per second is increased with the increase in the input size. It's again the same trend that we observed in the efficiency graph in figure 4.8. The percentage of increase in efficiency is decreasing in the experiments with higher number of records and the efficiency will become stable.

Details of Experiments

Input Records	Output Records	Elapsed Time (s)	No. of Maps	No. of Reducers
2098	2106	75	2	2
4200	4219	81	3	3
6004	6044	84	4	4
7758	7799	87	5	5
9364	9385	91	6	6
10874	10901	96	7	7
12689	12722	100	8	8
14279	14307	104	9	9
15759	15788	108	10	10
17364	17421	110	11	11
18705	18755	115	12	12
20186	20226	120	13	13
21688	21745	127	14	14

Figure 4.10: Overview of Experiments for Synonym Identification

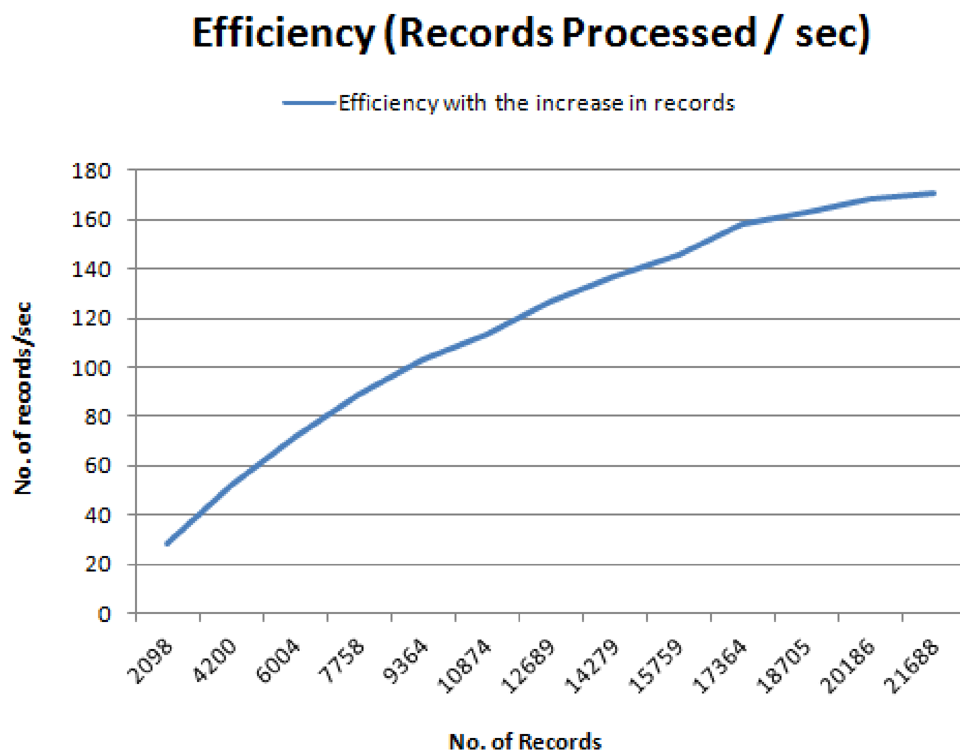

Figure 4.11: Efficiency Overview

The process of identifying synonyms will not reduce any records as all the identified synonyms are marked with a special text but not merged together, there might be a possibility that the overall records are increased. The graph in figure 4.12 shows the effect of this process on the total number of records. It can be seen that there is a negligible increase of records after this process.

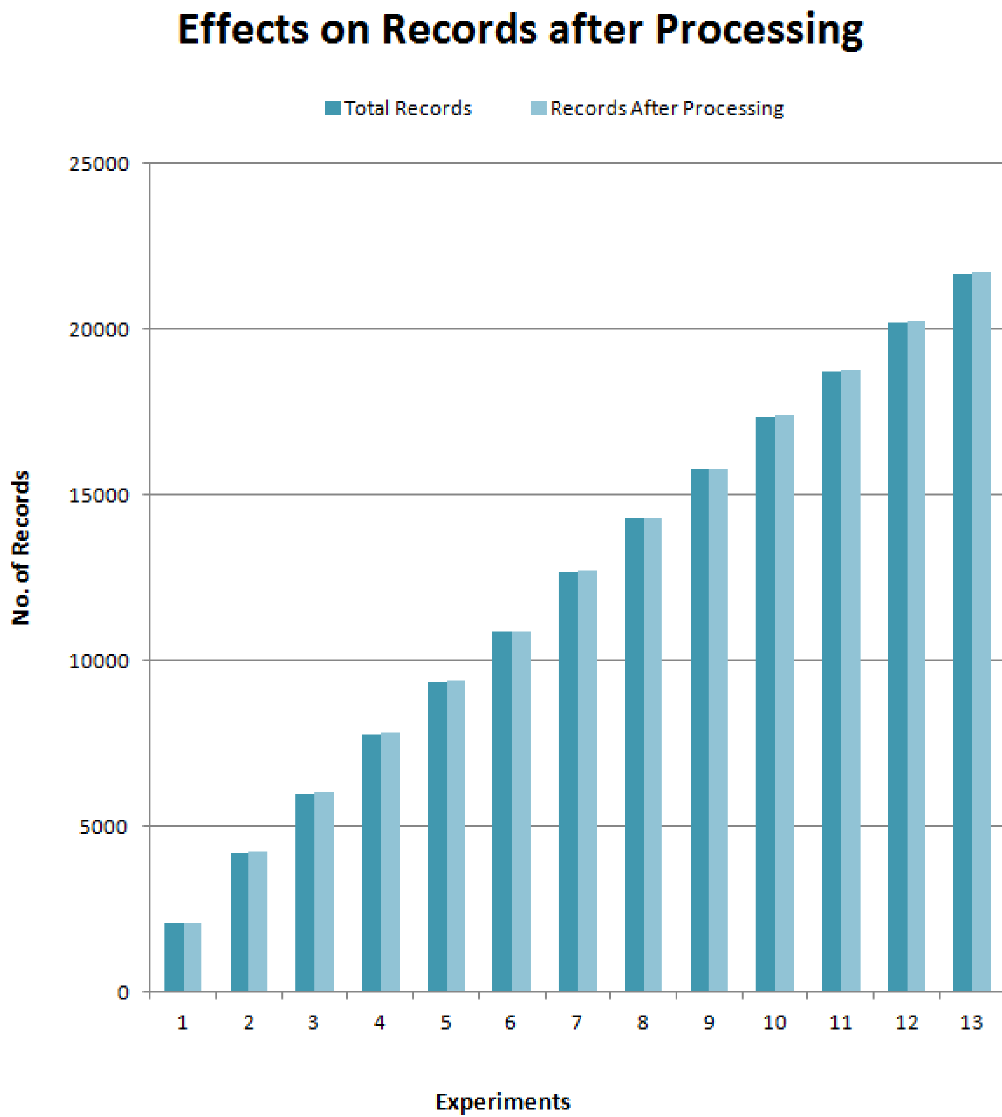

Figure 4.12: Overview of Records

4.7 Analysis

The experiments for all three processes showed that the algorithms are performing well with the increase in input records. The overall efficiency of the process is increased when executed on bigger data sets. The experiments also showed that the process is scalable as well.

In the initial experiments, approximately 143000 records are processed through the set similarity join algorithm in 51 seconds with 2 maps and reduce tasks each. During this process, 2220 concepts are identified. In the final experiment, approximately 3.6 Million records are processed in 103 seconds with 50 maps and 26 reduce tasks. The process results approximately 42500 concepts in total. The results clearly depicts that the efficiency of the process is increased and it also scales up very well.

Chapter 5

Related Work

A significant amount of work has been done in the domain of processing large data sets with Map-Reduce including the integration of data sets with set similarity joins as well. The contributions related to this project are explained below.

5.1 Efficient Parallel Set-similarity Joins Using MapReduce

In [13], the authors proposed a three stage approach for end to end set-similarity joins. The input is taken as set of records and output has been generated as set of joined records based on set-similarity condition.

The first stage is token ordering which is further divided into two different approaches i.e. Basic Token Ordering and One Phase Token Ordering. In Basic Token Ordering, the frequency of each token is calculated and then the records are sorted on the basis of frequency. The process is completed in two different Map-Reduce phases. To complete the same task in a single Map-Reduce job, One Phase Token Ordering approach is used; in which the sorting of records is explicitly done in memory. In the second stage of the process RID-Pairs are generated by calculating the prefixes of each record which are further used in the third stage for joining the records by using different techniques i.e. Record joins, R-S joins and Self joins. The overall process is explained in detail in figure 5.1.

The approach discussed in this paper involves quite a number of Map-Reduce jobs to achieve the desired result which needs more resources as well. Also, it measures the similarity on the basis of tokens and RID pairs which can only join the records syntactically but not semantically. Whereas, joining the records on the basis of semantic similarity is done in this thesis in the Synonym Identification process.

Figure 5.1: Three phase approach for Set Similarity Joins

5.2 Efficient Similarity Joins for Near Duplicate Detection

In [15], similarity joins for near duplicate data are discussed. The authors discussed prefix filtering methods and proposed an efficient positional filtering algorithm to find pairs of records having similarities up to a specific threshold value. The algorithm exploits the orders of the tokens for each data set. The tokens are then compared with each other and join algorithms like PP-Join and PP-Join+ are used for merging the similar data sets.

The paper focus on finding the near duplicate web pages, which is in fact a bit different domain and have different requirements of similarities as compared to the work done in this thesis.

5.3 Efficient Exact Set Similarity Joins

Microsoft Research proposed an efficient way of performing exact set similarity joins in [2], they proposed an algorithm in which two input collections are compared and the algorithm identifies all the pairs of sets that have high similarity. The approach used by them happens to be shared by the ones used in this thesis as well.

The first approach they used is the basic threshold set similarity joins algorithm which is similar in most of the work done related to this domain. The second approach is called Hamming set similarity join in which the data strings to be compared are partitioned into 3-grams and then a hamming distance is calculated between them. The approach resembles with the one used in this thesis i.e. Dice coefficient for finding the similarity between two strings. The third approach is to use Jaccard coefficient for measuring the similarity between then sets. In this approach, naive implementation of Jaccard coefficient is used for finding similarity between the sets.

The approach used for set similarity joins in this thesis is more optimized because we discussed several cases where the output of Jaccard coefficient is not enough for the declaration of two sets as similar with each other.

As discussed before, the size of the sets being compared is really important to be included in the computation. Therefore, we modified our approach and included size as one of the deciding factors for similarity.

5.4 MassJoin: A mapreduce-based method for scalable string similarity joins

In [6], the authors focused on string similarity joins and introduced a framework called 'Mass Join' which performs both character based and set based similarity functions. For set based similarity, the strings are partitioned into tokens (can be an n-gram or a word) and then similarity is calculated by using Jaccard, Cosine and Dice coefficients. For character based similarity, the approach is to calculate the 'Edit Distance' which is defined as minimum number of character operations to transform one string to another.

The Map-Reduce process has two main stages; the filter stage and the verification stage. In filter stage, candidate pairs are generated and if there is similarity in two pairs then they should share same signatures in the map phase i.e. same key so that they can routed to the same reducer to get merged eventually. In the verification stage, the candidate pairs generated from the initial stage are verified because two different strings may share multiple signatures and there is also a possibility of having duplicate candidate pairs as well. All these cases are handled in this stage.

Fixed values are used as thresholds for Jaccard coefficient during the evaluation phase, whereas in this thesis; it is also optimized and the threshold values are not fixed, they are changed with the variance in the size of the sets being compared.

5.5 Efficient Set Joins on Similarity Predicates

In [11], set similarity joins are performed on the basis of several measures like cosine similarity, Jaccard coefficient, intersection size etc. The basic similarity algorithms i.e. pair-count, word-groups etc are discussed, evaluated and optimized in this paper.

Improvements in the merging process are discussed as well. The set of lists to be merged are sorted on the basis of size and the largest set from the list is selected. The merging is done on the basis of threshold condition that a record should exist in at least one of the lists excluding the larger set. The primary focus of the work done in this paper is related to adapt memory intensive algorithms even when the available memory is limited. For this, data partitioning algorithms are used.

Although, the merging process is improved as compared to basic algorithms but still searching every record in the set of lists during the merge process can be a bit time and resource consuming. Also the step in which the sets are analyzed and decided to be merged together is not discussed in this paper.

5.6 Probase: A Probabilistic Taxonomy for Text Understanding

Microsoft Research proposed an efficient and probabilistic taxonomy construction algorithm in [14]. They called it 'Probase'. According to them, it is more comprehensive than any existing taxonomies and contains 2.7 million concepts. Probase uses probabilities to handle ambiguity and inconsistencies in the data it contains. Iterative extraction techniques are used for data extraction. The extraction part consists of two phases i.e. Syntactic and Semantic iteration.

The taxonomy is modeled as directed acyclic graph (DAG). There are two types of nodes i.e. concept and instance nodes. For the construction of probabilistic taxonomy two terms are introduced i.e. Plausibility and Typicality. Plausibility is defined as probability of a claim that it is true. Whereas, Typicality is the measure of how much an instance is related to a concept as compared to another instance.

The focus of this paper is to explain the information extraction process in detail. While the integration phase (scope of this thesis) is briefly explained. Map-Reduce framework is used for finding similarity between the concepts but the overall implementation details are not discussed here.

Chapter 6

Conclusion

In this project, we set out to introduce efficient data integration algorithms with distributed data processing techniques like Map-Reduce. The motivation behind the project was to improve the overall performance in processing large scale data sets; creating an overall efficient process of constructing taxonomy is one part of it. As discussed before, the overall efficiency not only depends on the data extraction process but as well as how efficiently data is integrated. We focused to improve the data integration algorithms.

Data integration involves finding similar data sets and merging them altogether to have unique information. To find similarity between data sets, we chose Jaccard coefficient to find out the similarity index between the data sets. We realized that the similarity index also depends on the size of the data sets in comparison. Therefore, we introduced two measures along with Jaccard coefficient to compute the similarity between the data sets and make this process more precise in declaring two data sets as similar.

Finding similarity can have several scenarios. We also tried to cater some realistic cases as well. Two of such cases are implemented in this project as well i.e. spelling difference identification and synonym identification.

Several experiments are performed in order to check the efficiency and scalability of the process with bigger data sets. As the input data is increased, the efficiency of the process also found to be increased until the allowed limit of the cluster for parallel execution was reached.

We found quite interesting results with out experiments on set similarity joins using Map-Reduce framework. We tested the maximum limit of cluster by increasing the number of records in the input. We checked the efficiency of our algorithm along with the scalability of the process as well.

Bibliography

[1] Comparison of similarity coefficients used for cluster analysis with amplified fragment length polymorphism markers in the silkworm, bombyx mori. *Journal of Insect Science*, 9(71):1–8, December 2009.

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. pages 918–929, 2006.

[3] Jairam Chandar. Join algorithms using map/reduce, university of edinburgh. 2010.

[4] C.L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275(0):314 – 347, 2014.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[6] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. pages 340–351, 2014.

[7] Lee Raymond Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, July.

[8] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. pages 938–948, 2010.

[9] Grzegorz Kondrak, Daniel Marcu, and Kevin Knight. Cognates can improve statistical translation models. In *HLT-NAACL*, 2003.

[10] Xueqing Liu, Yangqiu Song, Shixia Liu, and Haixun Wang. Automatic taxonomy construction from keywords. pages 1433–1441, 2012.

[11] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. pages 743–754, 2004.

[12] T. Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.*, 5:1–34, 1948.

[13] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. pages 495–506, 2010.

[14] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q. Zhu. Probase: A probabilistic taxonomy for text understanding. pages 481–492, 2012.

[15] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. pages 131–140, 2008.