A Thesis

entitled

Performance and Memory Enhanced Multiset Similarity Joins

by

Jahnavi Yalamanchili

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the
Master of Science Degree in Engineering

Dr. Robert C. Green II, Committee Chair

Dr. Vjiay Devabhaktuni, Committee Co-Chair

Dr. Kevin S. Xu, Committee Member

Dr. Amanda Bryant-Friedrich, Dean
College of Graduate Studies

The University of Toledo
December 2016
An Abstract of
Performance and Memory Enhanced Multiset Similarity Joins
by
Jahnavi Yalamanchili

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the
Master of Science Degree in Engineering
The University of Toledo
December 2016

The amount of data produced on a daily basis is growing at an exponential rate. One method of filtering through this data is the use of similarity joins, or methods that are used to identify similar data. Such algorithms are used for a variety of applications ranging from plagiarism detection to business analysis to DNA pattern prediction. These methods are typically time-consuming and computationally expensive.

This work proposes an efficient three-stage MapReduce algorithm named Adept Similarity Join (ASJ) for multisets. This kind of work is mostly used in data cleaning and duplicate detection applications. The main novelty in ASJ is to the integration of suffix filtering with positional filtering when performing similarity joins, in addition to incorporating prefix and size filtering. The proposed algorithm is compared to the state-of-the-art Strategic and Suave processing for performing similarity joins using MapReduce (SSS) algorithm, which it outperforms by lowering the number of redundant comparisons. This thesis also discusses memory used by an similarity join algorithms to process a dataset, an aspect which has been neglected in the literature.

Experimental results on a Twitter dataset and DBLP dataset using Hadoop demonstrate that the proposed ASJ algorithm provides about a 8% to 40% decrease in execution time and 100x reduction in memory usage compared to SSS.
I dedicate this work to my family
Acknowledgments

First and foremost, I would like to express my deep gratitude to my primary advisor and thesis committee chairman, Dr. Robert Green for all of the kind and enthusiastic motivation, support and guidance that he has provided me with over the course of my research.

I wish to express my very great appreciation to my co-advisor, Dr. Vijay Devabhaktuni for his thoughtful and patient guidance, encouragement and support over the course of my graduate studies.

I would like to offer my special thanks to Dr. Kevin Xu for his valuable and constructive suggestions during the period of my research work. His willingness to give his time so generously has been very much appreciated.

I thank Dr. Alam and for his kind and wise guidance throughout my Master’s program. I would also like to thank the EECS Department and Department of Psychology for the financial assistance. I thank Amazon for the Amazon Web Service (AWS) Education Research Grant award.

Most importantly, I would like to thank my friends and family for their continuous love and support.
Contents

Abstract iii
Acknowledgments v
Contents vi
List of Tables ix
List of Figures x
1 Introduction 1
2 Background 6
   2.1 Hadoop 6
      2.1.1 HDFS 6
      2.1.2 NameNode 7
      2.1.3 Secondary NameNode 9
   2.2 MapReduce 9
   2.3 Amazon Web Services 13
      2.3.1 Amazon S3 13
      2.3.2 Amazon EC2 15
      2.3.3 Amazon EMR 16
   2.4 Sets and Multisets 17
      2.4.1 Multiset Similarity Joins 19
2.5 Similarity Metrics ............................................. 20
  2.5.1 Character-based similarity metrics .................... 20
  2.5.2 Token-based similarity metrics ....................... 20
  2.5.3 Hamming Distance ..................................... 22

3 Related Work ................................................. 24
  3.1 Filtering Techniques ..................................... 24
  3.2 Similarity Join Algorithms ............................... 25
  3.3 Open Issues ............................................... 30

4 Adept Similarity Joins ....................................... 32
  4.1 Stage I ..................................................... 33
    4.1.1 Map phase ........................................... 33
    4.1.2 Reduce phase ....................................... 34
  4.2 Stage II ................................................... 35
    4.2.1 Map phase ........................................... 35
    4.2.2 Reduce phase ....................................... 36
  4.3 Stage III .................................................. 40
    4.3.1 Map phase ........................................... 40
    4.3.2 Reduce phase ....................................... 40
  4.4 AWS Configuration ....................................... 41
    4.4.1 Creating an Amazon S3 bucket ....................... 41
    4.4.2 Launching EMR cluster ............................. 42
    4.4.3 Processing data with a Custom JAR using Hadoop in AWS 49

5 Evaluation ...................................................... 50
  5.1 Twitter dataset .......................................... 50
  5.2 DBLP dataset ............................................. 55
# List of Tables

1. **Table of notation** .................................................. 39

2. Execution time in seconds taken by ASJ (± standard error over 5 runs) and SSS for varying thresholds .......................................................... 51

3. Physical Memory Used (GB) by ASJ and SSS for varying thresholds ................................................................................. 51

4. Time (sec) taken by Map and Reduce tasks in ASJ to complete stage I and stage II for varying thresholds .......................................................... 52

5. Output generated (GB) by ASJ and SSS for varying thresholds ....................................................................................... 52

6. Final stage output records produced by ASJ for varying thresholds ....................................................................................... 53

7. Candidate pairs generated (MB) by ASJ in different stages ....................................................................................... 54

8. Execution time in seconds taken by ASJ (± standard error over 5 runs) and SSS for DBLP dataset for varying thresholds ............................................................................. 55

9. Final stage output records produced by ASJ for varying thresholds ....................................................................................... 56

10. Execution time in seconds taken by ASJ (± standard error over 5 runs) for DBLP dataset for varying thresholds ............................................................................. 57

11. Final stage output records produced by ASJ for varying thresholds ....................................................................................... 58

12. Output bytes generated by ASJ in all stages ....................................................................................... 58

13. Size of candidate pairs generated (KB) by ASJ in different stages ....................................................................................... 59
# List of Figures

1-1 Example for data cleaning. .................................................. 2

2-1 Hadoop. .............................................................................. 8

2-2 MapReduce framework. ......................................................... 10

2-3 Diagrammatic representation of a set. ................................. 17

4-1 The three stages in ASJ. ....................................................... 33

4-2 Flowchart for stage I. .......................................................... 34

4-3 Flowchart for stage II. .......................................................... 37

4-4 Flowchart for stage III. .......................................................... 41

5-1 Total output records produced by ASJ for Twitter dataset in all stages. .......................................................... 53

5-2 Number of output records produced by ASJ in all stages. ....... 54

5-3 Comparison of output produced by ASJ and SSS for DBLP dataset in all stages (KB). .......................................................... 56

5-4 Candidate pairs generated by ASJ for DBLP dataset in each stage. ... 57

5-5 Comparison of candidate pairs (KB) generated produced by ASJ for DBLP dataset in all stages. ................................. 59

5-6 One of the outputs produced by ASJ for DBLP dataset. ............... 59

5-7 DBLP dataset for first output record. ...................................... 60

5-8 DBLP dataset for first output record. ...................................... 60

5-9 DBLP dataset for fifth output record. .................................... 60

5-10 DBLP dataset for fifth output record. ................................... 61
Chapter 1

Introduction

The amount of data generated on a daily basis has been increasing at an exponential rate. By 2020, the digital universe is predicted to grow by a factor of 300 from that of 2005 [1]. It is also estimated that by 2020, if analyzed, as much as 33% of the data might be of some value [1]. This massive data is created from many sectors including health, business, banking, manufacturing, retail, and social media. This “Big Data”, as it has come to be known, poses many challenges, chief of which is extracting meaningful value from such huge amounts of data. Distributed processing is usually employed to process very large datasets like this, with the end goal of extracting useful information.

The Apache Hadoop system [2] and MapReduce framework [3] are two prominent tools for such an approach. One of the key operations that are commonly performed when analyzing such data is a similarity join [4,5], a process which helps in duplicate detection [6,8], data cleaning [9], web-crawling [8], document clustering [10], social network mining [11,13], record linkage [14,15], query refinement [16], and plagiarism detection [17,18], among other applications.

Data collection is the process of gathering information, qualitative or quantitative, from surveys, interviews, focus groups or action research. It is an important part of business analysis to make critical decisions and predictions for future. Discrepancy in
**Figure 1-1: Example for data cleaning.**

The data collected is expected since a real-world object or element can be represented more than once and in more than one form due to manual errors like spellings. For example, names of authors can be repetitive since one entry could be just the first name and other might have the full name due to different representation in different systems. In the Fig.[1-1] it can be inferred that the second entry and fourth entry are the same, as well as the third entry and fifth entry. But the third entry does not have a middle name and the fifth entry has the full name. Similarly, Indiana Jones has two entries where one has the middle name and the other does not. Hence, data cleaning [19] becomes vital for the names to be joined and to have only one record for each name to make any subsequent decision based on that data.

Duplicate, or near duplicate, web pages are also causing a lot of concern as they require space to store the index, slow down the system, affect search results, or disturb users. Apart from detecting the near duplicate pages in web, there are documents that are stored in different places but in same form such as FAQs, files in mirror sites, and different versions of documents that are stored in the same place. There are also other causes for the existence of duplicate data: typographical errors, versioned files,
mirrored files, plagiarized documents, spam emails generated from the same source or template, etc.

Mining in social networking sites uses similarity joins to suggest possibly required items based upon a previous purchase or history. Many of the e-commerce websites such as Amazon, eBay, etc., employ data mining. Similarity joins are also used to show that two users have similar interest or that their profiles match. This is especially used in dating sites such as eHarmony or LinkedIn websites to show a possible connection.

Detecting such similarities generally requires a similarity join. A similarity join involves finding and combining pairs of records in a dataset that exhibit similarity greater than a certain threshold. The similarity function returns a value from 0 to 1. A greater similarity value implies that the elements are more similar and may be duplicates. If a similarity join is performed in an uninformed way, e.g., by computing similarity of all items and comparing with all other items, the method becomes computationally expensive and requires a vast amount of memory for processing [5]. If done intelligently, the same type of analysis can be performed in a computationally efficient manner.

Performing similarity joins on a large data is a challenging task since terabytes of data cannot be processed in the memory of one machine. For example, the GeneBank dataset consists of 100 billion bases and 100 million sequences, and an unknown gene has to be processed to find a similar match in these bases and sequences [20]. Applications with large datasets make use of clusters, and parallel algorithms are applied to work on this competently. Hence, division of labor is required. Because this is a pair-wise dealing, dividing the data to assure load balancing while reducing communication costs and keeping repetitiveness to a minimum is difficult. It is further complicated by the need to process various data sets with unbiased distribution and great dimensionality. Another challenge is to design a program so that the filters can cut short the unique pairs without actually computing their similarity until the pairs
reach the final stage. Filters that can support more than one similarity function is an useful purpose.

An emerging field in this area is the analysis of the similarity of multiset \[21\]. Unlike a set, a multiset contains the frequency information of its elements whereas sets do not. For example, two records \((a, a, b, c, c)\) and \((a, b, b, c, c)\) have the same set representation \((a, b, c)\) but different multiset representations due to the different frequencies of \(a\) and \(b\). The concept of multiset is being explored due to its superiority in many applications including as duplicate detection \[22\]. The bag-of-words model \[23\] is another application of multisets that is frequently used in natural language processing \[24\], information retrieval \[25, 26\] and computer vision \[27\]. In a bag-of-words multiset, a sentence is converted into a list of words and ranked according to the global frequency. Analyzing multisets at scale is a difficult task because the filtering techniques for multisets have to be designed to be suitable for parallel and distributed systems like MapReduce.

Given the importance and wide applicability of multisets, this thesis proposes and evaluates an improved similarity join algorithm for multisets. The main contributions are as follows:

1. A three-stage Adept Similarity Join (ASJ) MapReduce algorithm for finding similarity joins in multisets using stage-wise filtering techniques to prune the candidate pairs generated is proposed.

2. Suffix filtering is incorporated in addition to positional filtering in ASJ and the importance of suffix filtering in reducing redundant comparisons is demonstrated;

3. It is demonstrated that ASJ provides significant performance improvements in multiset similarity joins, including a roughly 8% to 40% decrease in execution time and a 100x reduction in memory usage.
4. A full performance analysis of ASJ is conducted using Twitter dataset in comparison with SSS and results from DBLP dataset are also recorded.

The remainder of this thesis is structured as follows: Chapter 2 overviews the work that has been done in this area as well as topics relevant to the current work, Chapter 4 discusses the proposed algorithm, Chapter 5 presents results and analysis, and Chapter 6 concludes the paper.
Chapter 2

Background

This chapter reviews relevant technologies (Hadoop and MapReduce) and algorithms (multiset similarity joins).

2.1 Hadoop

Hadoop [2] is an open-source software product that was developed by the Apache Software Foundation for scalable, reliable, and distributed computing. It consists of a fault-tolerant data storage system, known as the Hadoop Distributed File System (HDFS), where input and output is stored, and a processing system called MapReduce. A typical Hadoop usage pattern can be put into three stages:

1. Loading data into HDFS;

2. Performing MapReduce operations; and

3. Retrieving results from HDFS.

2.1.1 HDFS

HDFS is a file system which has distributed data storage across several machines. Its key features are:
1. It has storage blocks of at least 64MB size;

2. The rate of processing is optimized and it is effective for large files but poor for small files;

3. There is a master NameNode that controls each storage node called DataNode;

4. HDFS handles disk failures by using replication and NameNode handles it; and

5. Each DataNode reports back to the NameNode and NameNode monitors the reports to make sure failures do not occur.

The key components of HDFS are DataNode, NameNode, and Secondary NameNode. DataNode, as the name represents, is the data holding node. Every DataNode sends a message every 3 seconds to indicate that it is still operational. If the NameNode does not receive a message from a particular DataNode for ten minutes, then the NameNode declares that DataNode to be out of service. It then initiates another DataNode that stores replication of data from the out of service DataNode. The DataNodes talk to each other to balance data and keep the replication rate high.

### 2.1.2 NameNode

The NameNode is the master node for all data nodes. It controls what data blocks make up a file and the DataNodes in which those blocks are stored in for the cluster. It maps the locations of blocks and DataNode by maintaining two memory tables, one for mapping of blocks to DataNode and the other is for DataNode to block number mapping and this is stored as a registry in NameNode. This information is located in an image file called fsimage. When a disk corruption of a particular block happens, the first table gets updated and when the DataNode is observed to be dead, then both tables get updated. The NameNode stores two data structures, the fsimage file and the edits log of changes to it. The fsimage, as previously mentioned, holds the key
parts of a file system, details of each file and the mapping of blocks that corresponds to each. If the fsimage file is lost, we have lots of blocks of data and no knowledge of where to send them to, which leaves it useless. NameNode reads the fsimage file during start up and is sustained and altered in memory and any changes are listed in the edits log throughout the NameNode run time.

When the system starts, NameNode enters into safe mode. Each DataNode sends a message to NameNode to notify their existence and also the data block report. NameNode verifies that each DataNode has sufficient replicas and then it creates blockmap from block reports. A point here to note is that replication does not take place in safe mode. After generating blockmaps, NameNode exits safe mode and then any required replication happens. Safe mode for NameNode is the phase where it gets everything ready to start the process and it does not allow any changes to file system or blocks.
2.1.3 Secondary NameNode

The Secondary NameNode is a function that keeps a copy of edit logs and the fsimage file. The file system data is replicated into local/remote storage by Secondary NameNode. The name, Secondary NameNode, is misleading since its role is to perform checkpoints (not replace or take the position of NameNode) and to connect to the data nodes in case of its failure. The main purpose of Secondary NameNode is to perform periodic check points. It periodically takes the current edits log file from NameNode and fsimage, joins them into a new image and uploads it back to the primary NameNode. If the NameNode fails, it can just be restarted instead of shutting down the DataNodes. Secondary NameNode is not compulsory in a Hadoop cluster. But the changes in the file system are not directly made into the fsimage file, the alterations are noted in the edits log file and after a periodic time, Secondary NameNode joins both the image file and edits log file to get the new image file with latest changes and this takes time. If a Secondary NameNode is present, it speeds up the process.

2.2 MapReduce

MapReduce is a reliable, clustered programming paradigm that allows for processing enormous datasets with a parallel, distributed, and fault tolerant algorithm. There is another concept of Hadoop framework called Data Locality, meaning that Hadoop attempts to maintain a high replication rate so that as much data as possible appears as local. In other words, the data is located where it is needed, leading to reduced computation times.

MapReduce initially divides work among multiple map tasks. The map tasks, which run concurrently, process the input data by dividing it into independent fragments of key/value pairs. The framework then sorts and partitions the output
of the map phase. The resultant data is passed to the reduce phase, where multiple reducers function in parallel. Data is distributed to these reducers using a partition function. The mathematical representation of the partition function is given in [3] as:

\[
\begin{align*}
\text{Map} &: (key0, value0) \to (key1, value1)^* \\
\text{Reduce} &: (key1, (value1)^*) \to (key2, value2)^*
\end{align*}
\]

If \( R \) is the number of partitions or reducers, \( \text{hash}(key)\%R \) is the default partitioning function, where \( \% \) denotes modulus. After partitioning, the key/value pairs are shuffled based on primary key and sent as input to the reduce phase, which uses multiple reducers (which also run simultaneously with each other). The reduce phase then combines the key/value tuples into smaller sets of tuples. The MapReduce framework is represented diagrammatically in Fig. 2-2. Phases of MapReduce can be described as Initialization, Map, Shuffle, Sort, and Reduce.

MapReduce framework essentially performs the following functions:

1. Distributed processing and coordination;

2. Breaking down jobs into smaller chunks called tasks. These tasks are assigned
and organized in the cluster;

3. It performs Data Locality, i.e., code is moved to where the data is;

4. Failures are expected more due to large amount of data, so failed tasks are automatically re-tried on other machines;

5. Shuffle and sort phases readjusts and moves the data between nodes without any outside interference.

The key components of MapReduce are JobTracker and TaskTracker. The essential purpose of JobTracker is resource management, i.e., managing the TaskTrackers, tracking resource availability and task life cycle management. TaskTracker has simple function of following JobTracker’s orders and updating it with its status periodically.

Steps involved in submitting a job execution are:

1. Client applications submit jobs to the JobTracker;

2. The JobTracker talks to the NameNode to determine the location of the data;

3. The JobTracker locates TaskTracker nodes with available slots at or near the data and it submits the work to the chosen TaskTracker nodes;

4. The TaskTracker nodes are monitored. If they do not submit heartbeat signals often enough, they are deemed to have failed and the work is scheduled on a different TaskTracker;

5. A TaskTracker will notify the JobTracker when a task fails and the JobTracker decides what to do then, i.e., it may resubmit the job elsewhere, it may mark that specific record as something to avoid, or it may even blacklist the Task-Tracker as unreliable; and

6. When the work is completed, the JobTracker updates its status.
A few notable facts about JobTracker and TaskTracker are that the JobTracker is crucial for the Hadoop MapReduce service and if it goes down, all running jobs are halted. The TaskTracker is pre-configured with the number of slots indicating the number of tasks it can accept. When the JobTracker tries to schedule a task, it looks for an empty slot in the TaskTracker running on the same server which hosts the DataNode where the data for that task resides. If not found, it looks for the machine in the same rack. There is no consideration of system load during this allocation. HDFS is rack aware in the sense that the NameNode and JobTracker obtain a list of rack IDs corresponding to each of the slave nodes (data nodes) and creates a mapping between the IP address and the rack ID. HDFS uses this knowledge to replicate data across different racks so that data is not lost in the event of a complete rack power outage or switch failure.

Hadoop also performs speculative execution. This means that if a machine is slow in the cluster and the map/reduce tasks running on this machine are holding up to the entire map/reduce phase, then it runs redundant jobs on other machines to process the same task, and whichever task gets completed first reports back to the job tracker and results from the same are carried forward into the next phase. The TaskTracker spawns different JVM processes to ensure that process failures do not bring down the TaskTracker. The TaskTracker keeps sending heartbeat messages to the JobTracker to say that it is alive and to keep it updated with the number of empty slots available for running more tasks. The JobTracker does some check pointing of its work in the file system. Whenever it starts up, it checks what was it up to till the last CP and resumes any incomplete jobs. Earlier, if the JobTracker went down, all the active job information used to get lost. The status and information about the JobTracker and the TaskTracker are available on a web interface.

There are three modes of Hadoop. They are:

1. Local standalone mode: NameNode, DataNode, JobTracker, and TaskTracker,
run as a single Java process. In this case, it runs like a Java application, i.e., services do not have to be started in particular.

2. A pseudo-distributed mode: NameNode, DataNode, JobTracker, and TaskTracker run as separate Java processes but on a single host. Here, every component run as a part of a single machine.

3. Fully distributed mode: In this mode, Hadoop is spread across multiple machines, some of which will be general-purpose workers and others will be dedicated hosts for components, such as NameNode and JobTracker. This will be like an ideal cluster, the way we use in production environments.

2.3 Amazon Web Services

Amazon offers IT infrastructure as web services that is called as Cloud computing. It includes Simple Storage Service (S3), Elastic Compute Cloud (EC2), Elastic MapReduce (EMR) of many other services.

2.3.1 Amazon S3

S3 is a payable service that provides users with safe, long-lasting, and highly scalable cloud storage. Using S3, any content at any time can be retrieved or uploaded with a web interface. To start up, there is no minimum fee and no setup cost except for the storage space one uses.

Amazon S3 offers a range of storage classes designed for different uses: Amazon S3 Standard for general-purpose storage of frequently accessed data, Amazon S3 Standard - Infrequent Access (Standard - IA) for long-lived, but less frequently accessed data, and Amazon Glacier for long-term archive. Amazon S3 also offers configurable life cycle policies for managing your data throughout its life cycle. Once a policy is
set, the data will automatically migrate to the most appropriate storage class without any changes to the application [28]. Amazon S3 can be used just for storing purposes or the other services like Amazon EMR or EC2 can be utilized on the stored content. Some of the live examples making use of Amazon S3 are: Netflix delivers billions of hours of content from S3, SmugMug stores billions of photos and images on S3, Airbnb handles over 10TB of user images on Amazon S3.

The benefits of Amazon S3 are that it is designed for 99.999999999% durability to store important data since it stores in buckets across different locations and machines. Amazon S3 allows storage of large amounts of data at a very low cost. Using life cycle management, policies can be set to automatically migrate the data to Standard - Infrequent Access and Amazon Glacier as it ages to further reduce costs. Amazon S3 supports data transfer over SSL and automatic encryption of your data once it is uploaded. Using AWS Identity and Access Management (IAM), it can be configured, and object permissions and control access to the data can be managed. It can also be configured, manage object permissions, and control the access to data. With Amazon S3, one can scale up to any number when needed and down when it is not required. It can also send event notifications when objects are uploaded to S3 so that any processing can be done when they are available to use. S3 ensures high performance by utilizing maximum network throughput and flexibility. Amazon S3 is integrated with Amazon CloudFront, a content delivery web service that distributes content to users with low abeyance, and high data transfer speeds. It also allows transfer of huge amounts of data using Amazon S3 Transfer Acceleration which combines innovative software, protocol optimizations, and AWS edge infrastructure to accelerate data transfers up to 300% on existing infrastructure [28].
2.3.2 Amazon EC2

Amazon EC2 is a web service that arranges scalable computer capacity in the cloud and makes web-scale cloud computing easy to access for users.

This service allows to configure smoothly and also provides the control of Amazon’s computing environment. Amazon EC2 minimizes the time required to obtain and boot new server instances, which allows for scalability, both up and down as per the computing requirements. EC2 charges only for the capacity that is actually used. It also provides the necessary tools to build failure resilient applications and separate them from common failure systems. The advantages of choosing EC2 is that it enables one to increase or decrease computing capacity in minutes, not hours or days and hundreds or thousands of server instances can be delegated simultaneously. These instances can be completely controlled and root access to each one and IAM access can be obtained for other users. It can also be stopped and rebooted the instance while holding the data on that instance. It also provides flexible cloud hosting services such as instance types, operating systems, and software packages. EC2 gives the option of selecting a configuration of memory, CPU, instance storage, and the boot partition size that is useful for the choice of operating system and usage.

This is also designed to work with other Amazon web services like Amazon S3, EMR, RDS, etc., to give a solution for all the computing, processing and analysing needs for various applications. Amazon EC2 gives 99.95% reliability for each region where replacement instances are rapidly commissioned. It is inexpensive and easy to start for new users to get used to this. On top of this, Amazon EC2 works closely with Amazon Virtual Private Clouds (VPC) to provide security and powerful networking system.

Examples of this web service are: TimeInc uses AWS and migrates 5 data centre, Nasa JPL went from 15 days to a few hours on the same task using AWS, Animoto
used AWS to be able to handle 750,000 new users in 3 days.

## 2.3.3 Amazon EMR

Elastic MapReduce is also a service provided by Amazon to rapidly and cost-effectively process huge loads of data. EMR disentangles big data processing and provides a managed Hadoop framework through which vast data can be processed with scalability, ease and cost-effectiveness by using EC2 instances and storing the results in S3. Other distributed frameworks such as Apache Spark and Presto are also available in Amazon EMR.

It securely employs many big data cases such as log analysis, web indexing, data warehousing, machine learning, financial analysis, scientific simulation, and bioinformatics. Amazon EMR can also be used for analysing click stream data to group the users and understand user interests better. Razorfish uses EMR for click-stream analysis. The huge logs of data created by several web and mobile applications can also be processed using Amazon EMR. It also ameliorates users by converting peta Bytes of unstructured or semi-structured data into valuable observations. It can also be used on 1000 Genomes project, that is hosted on S3 for free, where this large data can be processed swiftly and smoothly.

Moreover, Amazon EMR is elastic, i.e., it can accommodate thousands of instances to process data at any scale. The user need not worry about node provisioning, cluster setup, Hadoop configuration or cluster tuning, the user needs to only mention the number of nodes and EMR takes care of it all. EMR, by default, configures firewall settings and the user can choose to use it only by customer-managed keys or AWS Key Management Service.

Many companies make use of the benefits of this service. GumGum uses EMR with Spark for inventory forecasting, processing of click-stream analysis and ad hoc analysis of unstructured data. CrowdStrike uses EMR with Spark to process hundreds
of peta Bytes of data to determine any kind of malignant activity. Swipely analyzes millions of credit card transactions to gather beneficial observations. Expedia makes use of the available click-stream data from global network websites for it to analyze. Kik uses EMR with Hadoop Pig scripts to process log file data. The financial Industry Financial Authority uses EMR to create a flexible platform that can accustom to changing marketing dynamics. Channel 4 uses customer interaction data for video-on-demand service.

2.4 Sets and Multisets

A Set is a well-defined collection of distinct objects [29]. The objects that make up a set can be anything, from simple numbers to people and other sets. For example, a set of colors, $A = \{\text{blue, red, white}\}$ or a set of numbers $B = \{5, 7, 7, 9\}$. The set $B$ is exactly identical to the set $C = \{5, 7, 9\}$ because sets do not take repeated objects into consideration. Diagrammatic representation of a simple set is given in Fig.2-3. The red area in between two sets $A$ and $B$ is the intersection or common elements of
sets A and B. If there are no common elements, then \( A \cap B \) is said to be null.

Assume that there are three sets, \( A = (1, 2, 3) \), \( B = (2, 3, 4) \) and \( C = (1, 2) \). The main concepts of sets are discussed below. The union between these two sets will be: \( A \cup B = (1, 2, 3, 4) \). The intersection between the sets, A and B will be: \( A \cap B = (2, 3) \). If all the elements of C are also elements of A, then C is a subset of A. Here, all the elements of C are also the elements of A. Therefore, \( C \subset A \). C is called a proper subset of A if and only if C is a subset of A but C is not equal to A. In the above example, C is a proper subset of A. A power set of a set X is the set whose members are all possible subsets. The power set of C is \(( (1), (2), (1, 2), () ) \).

The concept of multiset can be defined with the following characteristics as seen in:

1. A multiset is the collection of elements in which specific elements may appear more than once;
2. Occurrences of a specific element in a multiset can be identical;
3. Each occurrence adds up to the size of the multiset;
4. The number of occurrences of a specific element gives the frequency of that element in a multiset and is a finite positive integer;
5. The number of unique elements in a multiset need not be finite;
6. A multiset is completely consummated if the elements belong to it and the frequency of each element is known.

The number of times an element is recurring is called its frequency or multiplicity. A multiset is called ‘regular’ or ‘constant’ and corresponds to an ordinary set if all of its elements occur with the same multiplicity.

Multisets are also applied in Mathematics, especially in the areas of fuzzy set theory and usefulness of multisets in the study of combinatorics and formal languages are
also discussed in the literature [32]. Mathematics of multisets also proves a potential tool to straighten up the laws of polynomial rings [22]. The work in [33] concentrates on the implementation of subset and multiset subset lattices to the study of numbers, sizes, and properties of chains and anti-chains. Employing multiset theory to the subject of invariant theory, ring theory and theory of partitions has also been studied [34]. Multisets are also used in developing a multiset algebra [35] and it provides a connection between eigen values/eigen vectors equations Max-plus and non negative real number systems.

Multisets are also used in many computer science applications [22]. Knuth introduces multisets into algorithms where values such as \( y^n \) are computed [36]. Multistes and permutations of multisets are applied in different search and sort problems. The study in [37] finds use of multisets to know how often an assumption is repeated in an argument or a discussion. The work in [38] also employs multisets in algebra in developing an automated theorem for relevant logics. The use of multisets is exploited in numbers to explain the definition and conjecture formation program in [39]. The application of multisets in Logic Programming is observed to have conquered the computational inefficiency problem which is very useful in dealing with real-life situations where multiple instances or repetitions are important. The interaction between objects of multiset can imitate the biological evolution leading to the possibility of DNA computing and programmable living machines. From a computational point of view, a multiset can be considered as a data structure which is unique from a set, allows repetitive objects, and can be ordered, allowing it to be used as an apt modelling tool for a large class of real-life phenomena [22].

### 2.4.1 Multiset Similarity Joins

A multiset similarity join is obtained by finding pairs of records of multisets with a similarity count higher than a given threshold. More formally stated, a multiset
similarity join uses a desired threshold limit $t$ to find the similarity between multisets from a collection of records, $R = (X_1, X_2, \ldots, X_N)$, using a similarity function, $Sim$, such that the similarity amidst two multisets is greater than or equal to the given threshold (i.e. $Sim(X_i, X_j) \geq t$).

Within this process, a key operation is the calculation of similarity between multisets. Ruzicka similarity, cosine similarity and overlap similarity are some of the similarity functions commonly used in the literature and are defined in [40].

2.5 Similarity Metrics

Similarity metrics can be broadly classified into two types. They are character-based similarity metrics and token-based similarity metrics.

2.5.1 Character-based similarity metrics

These measurements evaluate the similitude between two strings in view of character changes. They are fit for catching typographical mistakes. One delegate character-based metric is edit distance. The edit distance [41] between two strings is the base number of alter operations required to change one string to the other, where the edit operations incorporate inclusion, cancellation, and substitution. There are likewise other separation measurements, for example, Euclidean separation [42], Manhattan separation [43]. Each of these has their favorable circumstances and disservices. As indicated by those favorable circumstances, they are utilized in specific applications.

2.5.2 Token-based similarity metrics

These metrics first change strings into sets of tokens and after that utilize the set-based similarity metrics to evaluate their similarity. The token-based measurements

20
are reasonable for long strings, e.g., archives. Two systems are generally used to change strings into sets: (1) tokenization and (2) q-grams. The previous one tokenizes strings based on unique characters, e.g., white-space characters. The last one uses a string’s sub-strings with length q to create the set, where the sub-string with length q is called a q-gram.

In this study, the token based similarity metric of the Ruzicka similarity function (a generalization of Jaccard similarity to multisets) that is shown in (2.3) is adopted.

\[ Sim(X_i, X_j) = \frac{|X_i \cap X_j|}{|X_i \cup X_j|} \]  

(2.3)

In a collection of multiset records, \( R = (X_1, X_2, \ldots, X_N) \), the \( i^{th} \) multiset is given by \( X_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n}) \) where the \( n^{th} \) element of \( X_i \) is given by \( x_{i,n} = (d_n, f_i(d_n)) \). \( d_n \) represents the \( n^{th} \) data element, and \( f_i(d_n) \) denotes the frequency of \( n^{th} \) data element in the \( i^{th} \) multiset. A multiset union is taken as the maximum frequency of data element in a multiset pair, i.e.

\[ |X_i \cup X_j| = \sum_{n \geq 1} \max(f_i(d_n), f_j(d_n)) \]  

(2.4)

Similarly, intersection is the minimum frequency of data elements in a multiset pair, i.e.

\[ |X_i \cap X_j| = \sum_{n \geq 1} \min(f_i(d_n), f_j(d_n)) \]  

(2.5)

There are also other type of similarity functions such as Cosine, Overlap and Dice functions. Cosine similarity function is obtained by using Euclidean product formula. The cosine of two non zero vectors is given by \( 2.6 \)

\[ a \cdot b = ||a|| \ ||b|| \cos \theta \]  

(2.6)
From [2.6],

$$\cos \theta = \frac{X_i \cdot X_j}{||X_i|| \ ||X_j||} \quad (2.7)$$

$$\cos(\theta) = \frac{\sum X_i X_j}{\sqrt{|X_i| \cdot \sqrt{|X_j|}} \quad (2.8)$$

The overlap similarity function is defined by the Eq [2.9]. If \( X_i \) is a subset of \( X_j \) or converse, then the overlap similarity is one. Overlap coefficient is also called as Szymkiewicz-Simpson coefficient.

$$\text{overlap}(X_i, X_j) = \frac{X_i \cap X_j}{\text{min}(|X_i|, |X_j|)} \quad (2.9)$$

The dice similarity coefficient (CSC) or Sorensen-Dice index is used to compare similarity between two strings. Sorensen's formula can be given as:

$$\text{Dice}(X_i, X_j) = \frac{2|X_i \cap X_j|}{|X_i| + |X_j|} \quad (2.10)$$

The function ranges from zero to one. Dice similarity function is considered as a semi-metric version of Jaccard index.

### 2.5.3 Hamming Distance

The number of bits which differ between two binary strings is called as Hamming Distance. In other words, the minimum number of substitutions required for two strings to be equal is the Hamming Distance[41]. For example, Hamming Distance between

1. California and Caltfozonia is 2.

2. 101101 and 100101 is 1
Hamming Distance satisfies the triangle inequality, which is, sum of two sides in a triangle is greater than or equal to the third side,

\[ z \leq x + y \quad (2.11) \]

Python and C both have functions for hamming distance represented by `hammingDistance()` and `hamming_distance()`, respectively. In this work, Hamming Distance is used in suffix filtering of stage II. The work in [42] also proposes a Hamming Distance metric learning for large-scale multimedia applications to transform high-dimensional data to binary codes that preserve semantic similarity.
Chapter 3

Related Work

The current literature on similarity joins focuses on improving computational efficiency by using filtering techniques. Some of the algorithms and techniques for finding similarity joins have been developed for sets, vectors, and multisets. These techniques are discussed in this section.

3.1 Filtering Techniques

Count filtering is proposed by [43]. The basic idea is that if two strings are similar, their signatures must share at least $T$ common signatures. In other words, if the number of shared signatures between two strings is smaller than $T$, the string pair can be pruned. GramCount utilizes this property to support edit distance. ListMerger extends GramCount to support token-based metrics in [44, 45]. GramCount uses the concept of q-grams and generates q grams for each string and takes them as signatures. ListMerger employs inverted index using signatures to generate candidate pairs and the pairs that have lower threshold than $T$ are pruned.

Length filtering is proposed in [43]. The concept of length filtering is that if two strings are similar, their length difference cannot be large. To utilize this concept, strings can be divided into several groups.
3.2 Similarity Join Algorithms

The Adaptjoin algorithm [46] aims to improve prefix filtering for all similarity functions. It proposes that if there are \( l \) more tokens in the set that are joined to the prefix, two similar strings should share at least \( l + 1 \) signatures and this is called as \( l \)-prefix scheme. Hence, capability of filtering is enhanced but at the cost of more filtering.

The VChunk algorithm [47] is specifically applied to the edit distance concept. The signatures of this algorithm are several-length chunks, called vchunk. The key part of the signature generation is the Chunk Boundary Dictionary (CBD) which is a set of rules used to split the strings into chunks. VChunk proposes the tail-restricted CBD which is a subset of all the CBDs and proves that if using a tail-restricted CBD to split strings, any edit operation will destroy at most 2 chunks. Thus the gram-based technique can be applied to the chunks split by CBDs.

The PartEnum algorithm [19] is designed especially for the hamming distance algorithms. The signature of PartEnum is "partition" and "enumeration". It is based on two parameters: N1 and N2. In the first level, it partitions into N1 strings based on the threshold. In the second level, all possible enumeration signatures are generated.

The main purpose of FastSS [48] is for edit distance, that uses neighborhood-based method. The signature of this algorithm is neighbors. The overall concept is that if two strings are similar, their neighbors must overlap. Then it employs the filter-verification framework to join the results. But since it’s signatures are strings, it is not effective for long strings.

The Qchunk algorithm [49] is based on two types of signatures: q-grams and q-chunks. It uses two ways to discard dissimilar pairs. First is to index q-grams and use q-chunks to generate candidates. The parts of q-grams are q-chunks.
The PassJoin Algorithm [50] utilizes a partition-based framework. For Jaccard, Cosine and Dice PassJoin converts it into Edit Distance. For example, let us consider Jaccard. The flow goes like this: PassJoin sorts all the tokens in the set based on the global ordering, then it converts Jaccard threshold to Overlap and then to Edit Distance [51].

TrieJoin calculation [52] implements a trie structure to figure the likeness join result specifically. Each trie node is connected with a character and the way from the root to a leaf node relates to a string. Two strings with a typical prefix will share a basic predecessor. TrieJoin depends on a critical idea "active nodes". A node is called an active node for string s if the alter separation amongst s and the string w.r.t. the node is not bigger than a given limit. For self join, TrieJoin first forms a single trie structure and processes all the active nodes of the leaf nodes. At that point given a leaf node, the strings comparing to active nodes must be like the string relating to the leaf group. Consequently, TrieJoin, without much of a stretch, can acquire the join result based on active nodes of leaf nodes.

M-Tree is also a tree based index structure. The M-Tree algorithm [53] is based on the idea of triangle inequality to prune dissimilar pairs and it is build to support metric space similarity functions.

Locality Sensitive Hashing (LSH) [54] is a universally embraced technique for similarity search in large documents such as databases and it reduces the dimensionality in high-dimensional data. LSH uses hashing of elements so that similar ones reach the same bucket [55]. It is much similar to data clustering and nearest neighbor search. There are two methods used by hashing-based approximate nearest neighbor search algorithms. They are: locality sensitive hashing which are data-independent methods and locality preserving hashing (LPH) which are data-dependant methods. LSH has been found to have applied in various problem domain such as near-duplicate detection [56,57], hierarchical clustering [58], genome-wide association study [59] and
audio fingerprint. [60] created an anti-spam LSH algorithm and the goal is to generate a hash digest of an email message such that pieces if two similar messages are similar to each other. LSH is typically used for approximate similarity joins, whereas the focus of this paper is on exact similarity joins.

A SSJoin operator for filtering (prefix filtering) is initially developed for sets in [9] and is later improved to be implemented in vectors and multisets. There are several enhancements on the basic prefix-filtering scheme. The work in [9] is an RDBMS-based implementation that exploits the set-based and group-base processing mechanism by the RDBMS i.e., it implements the prefix filtering method on top of a commercial database system, while [61] further improves the method by utilizing several other filtering techniques in candidate generation phase and verification phase. An extension to SSJoin is exact-set similarity joins [19] where a signature-based technique called PartEnum is proposed. This algorithm assessed similarity for pairs that share at least one common signature with Hamming distance less than a certain predefined value. The study in [62] proposes a different inverted signature-based hash table (ISH) algorithm that combines inverted index based approach and signature based approach. It is applied for membership checking problem. According to this paper, it can filter out sub-strings that cannot match with any dictionary member, and then it verifies the rest of the sub-strings against the dictionary. But this method is also approximate method.

The All-pair [61] similarity join is an algorithm for sets and vectors. It applies an inverted list based approach and appropriately exploits the ordering of vectors. Size filtering for sets is implemented in [63]. The all-pairs similarity search problem is a generalization of the nearest neighbor problem in which the goal is to find the nearest neighbors of a given point query. Top-k similarity join [64] can be used when the threshold limit of similarity join is unknown. It employs the prefix filtering principle and tight upper bounding of similarity values by utilizing the concept of
monotonicity. It returns the pairs whose similarity is highest among all possible pairs. This technique is most applied in IR community to get the pairs classified in the order of their similarity.

PPJoin+ [6] proposed prefix filtering along with positional and suffix filtering for sets and was developed to detect near duplicate web pages. The work in [65] presents an extensive survey of techniques used for duplicate detection in database records.

Advances have also been made to perform similarity joins to multisets in an efficient manner. The V-SMART [66] algorithm is a two-stage method for multisets, sets and vectors. The first stage computes and joins the partial results, and the second stage computes the similarity for all candidate pairs. This algorithm revolves around unilateral partial results, Uni(M_i), and conjunctive partial results for each candidate pair. The first unilateral class has functions whose partial results can be computed using a scan on the elements in only one multiset, either U(M_i) or U(M_j). Unilateral functions consistently either f_{i,k} or f_{j,k}. Scanning only the elements in U(M_i), instead of the entire A, and adding the frequencies of elements in U(M_i) yields the size of M_i. Conjunctive functions are performed to find the intersection pairs of the two multisets and \(|M_i \cap M_j|\) is calculated. It also uses disjunctive functions which calculates the partial results by utilizing the absolute of the difference. It proposes three algorithms in the joining phase, the Online-Aggregation algorithm, the Lookup algorithm and the Sharding algorithm in the joining phase. Lookup step computes Uni(M_i) for each M_i. The output of the reducers are files mapping each Mi to its Uni(M_i). The first joining phase joins Uni(M_i) to all the elements in U(M_i) and this Look up joining phase seems to suffer with limited scalability. Sharding algorithm is the hybrid of Online Aggregation and Lookup algorithms. The second similarity phase builds the inverted index, and computes the similarity between all candidate pairs. This work is applied to detect similar IPs and cookies. Although V-SMART computes similarities in simple and clear way, the lack of filtering techniques makes
the algorithm time-consuming\textsuperscript{[67]}. 

VCL \textsuperscript{[13]} is a MapReduce version on PPJoin+ for set similarity join and VCL can also be used for vectors. It is proposed in three stages. The first stage computes data statistics to generate signatures which are used in later stages. Second stage extracts the record IDs and value so that the pairs sharing a signature go to common reducer. The reducers then computes the similarity of values and generate similar records. Stage three generates actual pairs of joined records by using the list of RID pairs from second stage. Two approaches are listed for the third stage of joining data. One is the Basic Record Join (BRJ) and One-Phase Record Join (OPRJ). It also discusses self-join and R-S join cases. VCL algorithm is a milestone research since the algorithm integrated filtering techniques with MapReduce framework, but the algorithm’s data replication diminished its performance.

The work in \textsuperscript{[67]} implemented a prefix filtering technique using MapReduce. It proposes a two stage algorithm in which it implements prefix filtering to reduce the candidate pairs, and second stage calculates similarities from the candidate pairs generated in the first stage. BJoin algorithm is also proposed to enforce prefix filtering in stage one. The proposed algorithm consists of two stages. The first stage filters out dissimilar pairs with prefix algorithm and second stage joins candidate pairs of the first stage and produces similar pairs that exceed a threshold. \textsuperscript{[67]} used cosine similarity function. It performed better than V-SMART in it’s work but it suffers with poor scalability.

SSJ-2R \textsuperscript{[68]} also implements prefix filtering with MapReduce. Two algorithms were proposed SSJ-2 and SSJ-2R. SSJ-2 implements prefix filtering technique while SSJ-2R uses a remainder file, along with performing prefix filtering technique, to distribute data that is likely to be used by every reducer. It generates its candidate pairs by using inverted index and filters by using prefix filtering. Signature based scheming is employed to calculate the similar records. SSJ-2R was applied to a
sample of the WT10g TREC Web Corpus.

SSS [69] evolved multiset similarity joins by enforcing positional filtering, prefix filtering, and size filtering. It is proposed in three stages. But the main drawback is that SSS inputs the *whole multiset* as one of the multiple mappers, after prefix filtering, in the second and third stages. This causes redundant comparisons which in turn leads to network overflow and memory and I/O inefficiency. The proposed approach builds on SSS but does *not* require the whole multiset, which results in significantly less memory and I/O requirements along with a moderate improvement in computation time.

### 3.3 Open Issues

Generally speaking, memory usage has not been given enough attention in the literature. Many of the proposed algorithms suffer from poor scalability or redundant comparisons. A lot of network and memory congestion is associated with those comparisons. Work has been done to implement prefix filtering alone or with a size filtering technique as applied to multisets, but these studies were limited to a few filtering methods which make for inefficient pruning methods. The proposed ASJ approach introduces suffix filtering to multisets, a technique that filters candidate pairs based on Hamming distance. The suffix filtering technique is included in the same stage as the positional filtering technique so that unnecessary candidate pairs are pruned in that stage rather than being passed on to the next stage, saving memory and time. In the final stage, similarity between obtained pairs is calculated by using the Ruzicka similarity function. The processing of Twitter dataset is done on Amazon Web Services using Amazon EMR, S3 and EC2 showing the exact memory and time used up by each of the processes.

It should be noted that the suffix filtering implemented in ASJ is motivated by the
suffix filtering proposed in [6]. The Prefix filtering used in this algorithm is initially developed for sets in [9], which is improved and implemented in multisets in this study.
Chapter 4

Adept Similarity Joins

This chapter proposes the ASJ algorithm, which uses a three stage approach. Prior to the first stage, the data is pre-processed by ordering the elements based on their global frequencies. Candidate pairs are generated in the map phase of Stage I when the prefix elements of multisets are indexed. In the reduce phase, the candidate pairs that have the same prefix are grouped and size filtering is performed to scale down the pairs. In Stage II, an overlap based positional filtering technique is applied to the candidate pairs, and if a candidate pair survives this positional filtering technique, suffix filtering is applied. In the final Stage, candidate pairs are obtained by finding the similarity between multisets using the Ruzicka similarity function. The stages and filtering techniques of ASJ are pictured sequentially in Fig. 4-1. The remainder of this section will detail the stages of the proposed algorithm.
4.1 Stage I

4.1.1 Map phase

The Stage I map phase retrieves the pre-processed input data. From the prefix filtering technique [9], if two multisets are similar, they should have a non-zero intersection in their prefixes. Prefix filtering is applied on multisets by calculating the prefix size \( P_i \) using (4.1).

\[
P_i = |X_i| - (t \times |X_i|) + 1\]  
(4.1)

Since prefix filtering itself is not sufficient to generate candidate pairs, an inverted index is used in this phase. By applying an inverted index, the data \( d_e \) will be sent as the key, and other information such as multiset ID or MID \( X_i \), size \(|X_i|\), and the frequency of \( d_e \) in \( X_i \) \( (f_i(d_e)) \) are sent as the value of each mapper. This means that the multisets that share the same element are merged as possible candidate pairs. The output of Stage I map phase is given below.

\[
Key : d_e \]  
(4.2)

\[
Value : X_i, |X_i|, f_i(d_e) \]  
(4.3)
4.1.2 Reduce phase

Each reducer groups the records obtained from the Stage I map phase with key \( (d_e) \). The prefix filtering technique is not effective with respect to the number of candidate pairs generated. Assuming \( n \) records have a common prefix data element, it generates \( n(n-1)/2 \) pairs, which is redundant. Hence, we use size filtering technique in the reduce phase for effective pruning. Each reducer gets the size of multisets that the candidate pair belongs to, and the following size filtering condition is applied:

\[
|X_j| \geq (t * |X_i|)
\]  

(4.4)

where \( X_j \) is the smaller multiset and \( X_i \) is the larger. Every reducer checks the above condition for each candidate pair, and if it holds true, it passes size filtering.
In any other case, it is recognized as not begin a potential candidate pair and is pruned. The MID pairs \((X_i, X_j)\), sizes of both the multisets \((|X_i|, |X_j|)\), and frequencies \((f_i(d_e), f_j(d_e))\) of the data in both multisets are sent as output of the reduce phase:

\[
\text{Key} : X_i, X_j \\
\text{Value} : |X_i|, |X_j|, f_i(d_e), f_j(d_e)
\]

The size filtering technique is very effective in filtering out the candidate pairs generated in the map phase, as observed in the results section. Stage I is represented as a flowchart in Fig 4-2.

4.2 Stage II

4.2.1 Map phase

The Stage II map phase reads the filtered candidate pairs to generate MID pair records. The output will be a key/value pair with the multiset IDs of candidate pairs as keys and sizes of multisets, frequencies and position information of data elements in multisets as values so that all the common MID pairs are partitioned to the same reducer, enabling us to perform positional and suffix filtering in the reduce phase.

Here, partitioning, grouping, and sorting are also performed. Custom partitioning considers the \(X_i\) as the primary key and \(X_j\) as the secondary key. The records are partitioned based on the primary key, \(X_i\) and the custom partitioning makes sure that pairs that have the same primary key reach the same reducer. Custom grouping ensures that each reducer vests to each primary key. Here, the positions of the data element, \(d_e\), in \(X_i\) and \(X_j\), \(pos(m_{i,e})\) and \(pos(m_{j,e})\) are also returned in the output.
The output of Stage II map phase is displayed below:

\[ Key: X_i, X_j \]  \hspace{1cm} (4.7)

\[ Value: |X_i|, |X_j|, f_i(d_e), f_j(d_e), pos(m_{i,e}), pos(m_{j,e}) \]  \hspace{1cm} (4.8)

### 4.2.2 Reduce phase

The MID pairs with the same primary key arrive in the same reducer instance and are sorted based on their secondary key, \( X_j \). The reduce phase implements positional filtering with suffix filtering to further reduce the candidate pairs. Positional filtering performs filtering on the last overlapping prefix element. The positional filtering principle is that the maximum overlap between two multisets, \((X_i, X_j)\), has to be greater than or equal to the equivalent overlap threshold. If \( w \) is the last token in the prefix, then the positional filtering principle can be represented as

\[
\text{Overlap}(w_{i,t}, w_{j,t}) + \min(|w_{i,r}|, |w_{j,r}|) \geq \alpha.
\]  \hspace{1cm} (4.9)

Since the positional filtering technique is applied to the last overlapping element, overlap between the left partitions will be the total prefix overlap \( \alpha_p \). Sizes of the right partitions in \( X_i \) and \( X_j \) can be calculated by deducting the current positions of the last overlapping element from the total multiset sizes. The equivalent overlap threshold is denoted by \( \alpha \) and is calculated by the formula below:

\[
\alpha = \frac{t}{1 + t}(|X_i| + |X_j|)
\]  \hspace{1cm} (4.10)

If the candidate pair passes positional filtering, it enters suffix filtering. Suffix filtering works on the principle that, for two multisets to be similar than the threshold, the suffix Hamming distance, \( H_s \), between two multisets must be less than the
maximum Hamming distance, $H_{max}$, i.e.

$$H_s < H_{max}. \quad (4.11)$$

In order to find the Hamming distance between two multisets, sizes of left and right partitions of the multiset’s suffix and the contribution from the partition element, $d_f$, are needed. Left and right partition sizes are calculated by using the suffix size and
Algorithm 1 Positional Filtering of Stage II Reduce Phase

1: \( \alpha_p \leftarrow \min(f_i, f_j) \)
2: \( \alpha \leftarrow \frac{1}{t+1}([|X_i| + |X_j|]) \)
3: \( \max_r \leftarrow \min\{[[|X_i| - pos(m_{i,e}) + f_i(d_e)], [|X_j| - pos(m_{j,e}) + f_j(d_e)]\}
4: if \( \alpha_p + \max_r \geq \alpha \) then
5: Pass the pair to suffix filtering (Algorithm 2)
6: else
7: Prune the pair
8: end if

the frequency. Suffix size can be calculated by using (4.12).

\[
|Suf(X_i)| = (t * |X_i| - 1) \quad (4.12)
\]

The partitioning data element, \((d_{e,i})\), in the suffix of \(X_i\) is searched for as an element greater than or equal to in the suffix of \(X_j\). If they are equal, then \(\min(f_i(d_e), f_j(d_e))\) becomes \(d_f\); otherwise, \(d_f\) is the sum of the elements' frequencies. Hence, the Hamming distance between two multisets can be calculated by using (4.13) with abs\((|L_i| - |L_j|)\) for the left partition of the suffix and abs\((|R_i| - |R_j|)\) for the right partition.

\[
H_s = abs(|L_i| - |L_j|) + abs(|R_i| - |R_j|) + d_f \quad (4.13)
\]

Now, the maximum Hamming distance is defined as

\[
H_{max} = |X_i| + |X_j| - 2\alpha - H_{p,min} \quad (4.14)
\]

where \(H_{p,min}\) is the minimum prefix Hamming distance that can be explained by the sum of the positions of the last overlapping elements in \(X_i\) and \(X_j\) subtracted from the total overlapped prefix. The output of the reduce phase is \((X_i, X_j)\) as key and information of sizes, frequencies and positions as value.

Stage II is shown as a flowchart in Fig. 4-3. For ease of understanding, a table of notation has been provided in Table 4.1. Pseudo code for the reduce phase positional
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_p$</td>
<td>Total prefix overlap</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Equivalent overlap threshold</td>
</tr>
<tr>
<td>$\text{max}_r$</td>
<td>Intersection of right partitions</td>
</tr>
<tr>
<td>$</td>
<td>L_i</td>
</tr>
<tr>
<td>$</td>
<td>\text{Suf}(X_i)</td>
</tr>
<tr>
<td>$d_{e,i}$</td>
<td>Partition element of the multiset $X_i$</td>
</tr>
<tr>
<td>$\text{abs}(</td>
<td>L_i</td>
</tr>
<tr>
<td>$\text{abs}(</td>
<td>R_i</td>
</tr>
<tr>
<td>$H_s$</td>
<td>Suffix Hamming distance between multisets</td>
</tr>
<tr>
<td>$H_{p,\text{min}}$</td>
<td>Minimum prefix Hamming distance</td>
</tr>
<tr>
<td>$H_{\text{max}}$</td>
<td>Maximum Hamming distance</td>
</tr>
</tbody>
</table>

filtering is shown in Algorithm 1 and the reduce phase suffix filtering is shown in Algorithm 2.
Algorithm 2 Suffix filtering of Stage II Reduce Phase

1: $|Suf(X_i)| \leftarrow (t|X_i|) - 1$
2: $L_i \leftarrow$ Position of $d_{e,i}$
3: if $d_{e,i} = d_{e,j}$ then
4:     $d_f \leftarrow \min(f_i(d_e), f_j(d_e))$
5: else
6:     $d_f \leftarrow f_i(d_e) + f_j(d_e)$
7: end if
8: $H_s \leftarrow \text{abs}(|L_i| - |L_j|) + \text{abs}(|R_i| - |R_j|) + d_f$
9: $H_{p,min} \leftarrow (\text{pos}(m_{i,l}) + f_i(d_l) + (\text{pos}(m_{j,l}) + f_j(d_l)) - 2\alpha_p$
10: $H_{\text{max}} \leftarrow |X_i| + |X_j| - 2\alpha - H_{\text{min}}$
11: if $H < H_{\text{max}}$ then
12:     Output the pair
13: else
14:     Prune the pair
15: end if

4.3 Stage III

4.3.1 Map phase

The MID pairs that survive positional and prefix filtering techniques reach the Stage III map phase. Intersection and union information of multisets that is required to calculate the similarity score is computed and sent to the reduce phase. Union and intersection between two multisets is calculated by using frequencies as given by (2.4), (2.5) respectively.

Key : $X_i, X_j$  

Value : Intersection, Union

4.3.2 Reduce phase

From the key/value pairs obtained in the map phase partitioning, grouping and sorting is performed and given to each reducer. In the final stage reduce phase, the similarity between two multisets is calculated using Ruzicka similarity function (2.3)
using intersection and union values obtained from map phase. The final output of the algorithm consists of the multiset pairs that are at least as similar based on the given threshold, and their similarity. MID pairs are generated as keys as given in (4.15), and their similarities are the values. Stage III is shown as a flowchart in Fig. 4-4.

4.4 AWS Configuration

4.4.1 Creating an Amazon S3 bucket

Amazon EMR can use Amazon S3 to store input data, log files, and output data. Amazon S3 refers to these storage locations as buckets. This section shows how to use the Amazon S3 console to create a bucket that stores cluster logs and output.
Creating a path for cluster logs is optional. When a cluster is launched using the console, if Amazon S3 log location is not specified, one is generated automatically.

To create an Amazon S3 bucket using the console:

1. Open the Amazon S3 console and choose Create Bucket. In the Create a Bucket dialog box:
   
   (a) Type a bucket name, such as myemrbucket. The bucket name should be globally unique. If the name that is typed is in use by another bucket, type a different name. Choose Create.
   
   (b) In the list, select the bucket name and choose Create Folder.
   
   (c) For Name, type output and then press Enter. This creates the following path for the output data: s3://myemrbucket/output.
   
   (d) For Name, type logs and then press Enter. This creates the following path for cluster logs: s3://myemrbucket/logs.

### 4.4.2 Launching EMR cluster

The following stride is to dispatch the Amazon EMR bunch. When a group is dispatched, Amazon EMR arranges Amazon EC2 occurrences (virtual servers) to play out the calculation. These examples are made utilizing an Amazon Machine Image (AMI) redone for Amazon EMR. The AMI has Hadoop and other information applications preloaded.

To launch an Amazon EMR group utilizing the console:

1. If there is no need of group logs, the logging choice may be impaired.

2. Open the Amazon EMR console at https://console.aws.amazon.com/elasticmapreduce/. Pick Create group.
3. On the Create Cluster page, in the Cluster Configuration segment, acknowledge the default alternatives. These choices are characterized in the accompanying points of interest:

(a) Cluster name: When a group is made, the default name is "My cluster." Anything can be used as a name for the group. The name is discretionary, and does not need to be interesting.

(b) Termination insurance: By default, groups made utilizing the console have end insurance empowered (set to Yes). Empowering end protection guarantees that the bunch does not close down because of mischance or mistake. Regularly, to secure long-running bunches, or to protect information, an application can be assured when it is being built so that errors can be troubleshooted that would have generally ended the group.

(c) Logging: By default, groups made utilizing the console have logging enabled. This alternative figures out if Amazon EMR composes point by point log information to Amazon S3. When this worth is set, Amazon EMR duplicates the log documents from the EC2 occurrences in the group to Amazon S3. Logging to Amazon S3 must be empowered when the group is made. Logging to Amazon S3 keeps the log records from being lost when the cluster closes and the EC2 occasions facilitating the bunch are ended. These logs are helpful for investigating purposes.

(d) Log folder S3 area: To store the Amazon EMR logs, it can be sorted or searched to the same S3 location; for instance, s3://myemrbucket/logs, Amazon EMR can be permitted to produce an Amazon S3 way. In the event that the written name of an envelope that does not exist in the can, it is created.

(e) Debugging: By default, when logging is empowered, investigating is like-
wise empowered. This choice makes an Amazon SQS trade to handle investigating messages.

(f) In the Tags segment, leave the choices clear. Labeling permits ordering of assets utilizing key-esteem sets. Labels on Amazon EMR groups are spread to the basic Amazon EC2 cases.

4. In the Software Configuration segment, acknowledge the default alternatives. These alternatives are characterized in the accompanying table.

(a) Hadoop appropriation: This alternative figures out which conveyance of Hadoop to keep running on the group. Naturally, the Amazon circulation of Hadoop is chosen, yet it can run one of a few MapR circulations

(b) AMI form: Amazon Elastic MapReduce (Amazon EMR) utilizes Amazon Machine Images (AMIs) to introduce the EC2 cases it dispatches to run a group. The AMIs contain the Linux working framework, Hadoop, and other programming used to run the group. These AMIs are particular to Amazon EMR and can be utilized just as a part of the connection of running a bunch. Of course, the most recent Hadoop 2.x AMI is chosen. One can likewise pick a specific Hadoop 2.x AMI or a specific Hadoop 1.x AMI from the rundown. The AMI picked decides the particular form of Hadoop and other applications, for example, Hive or Pig to keep running on the group. When the console is utilized to pick an AMI, deplored AMIs are not appeared in the rundown.

(c) Applications to be introduced: When the most recent Hadoop is picked, 2.x AMI, Hive, Pig, and Hue are introduced naturally. The applications introduced and the application forms change contingent upon the AMI selected.
(d) Additional applications: This alternative permits introduction of extra
applications, for example, Ganglia, Impala, HBase, and Hunk. When an
AMI is picked, applications not accessible on the AMI don’t show up in
the rundown.

5. In the File System Configuration segment, acknowledge the default alternatives
for EMRFS. EMRFS is an execution of HDFS which permits Amazon EMR
groups to store information on Amazon S3. The default alternatives for EMRFS
are characterized in the taking after focuses

(a) Server-side encryption: When utilizing the console to make a group, server-
side encryption is deselected as a matter of course. This alternative em-
powers Amazon S3 server-side encryption for EMRFS.

(b) Consistent perspective: When utilizing the console to make a bunch, steady
perspective is deselected naturally. This alternative empowers predictable
perspective for EMRFS. Whenever empowered, the EMRFS metadata
store, the quantity of retries, and the retry period has to be indicated.

6. In the Hardware Configuration area, for the Core EC2 example sort, pick
m3.xlarge and acknowledge the rest of the default choices. These choices are
characterized in the accompanying points.

Twenty is the default greatest number of hubs per AWS account. For example,
if there are two bunches, the aggregate number of hubs running for both groups
must be 20 or less. Surpassing this farthest point brings about group disap-
pointments. On the off chance that more than 20 hubs are required, you should
present a solicitation to expand Amazon EC2 occasion limit. Guarantee that
when more than breaking point increment is requested, it incorporates adequate
limit for any transitory, spontaneous increments in the necessities.
(a) Network: When utilizing the console to make a group, the default VPC is chosen consequently. If there are extra VPCs, an interchange VPC from the rundown can also be picked.

(b) EC2 Subnet: No inclination is chosen as a matter of course which permits Amazon EMR to pick an arbitrary subnet. Then again, a particular VPC subnet identifier from the rundown can be chosen.

(c) Master: Amazon EMR groups must contain 1 expert group. The expert hub is contained in an expert occasion group. EC2 occurrence sort decides the kind of virtual server used to dispatch the Amazon EMR expert hub. The occurrence sort picked decides the virtual registering environment for the hub: handling power, stockpiling limit, memory, et cetera.

The default expert occurrence sort for Hadoop 2.x is m3.xlarge. This imposition sort is reasonable for testing, advancement, and light workloads. By default, Count is set to 1 for the expert hub. Right now, there is as it were one expert hub for every group. Demand spot is unchecked as a matter of course. This alternative indicates whether to run expert hubs on Spot Instances.

(d) Core: The center occasions run Hadoop errands and store information utilizing the Hadoop Distributed File System (HDFS). The group must contain at slightest 1 center hub. Center hubs are contained in a center case bunch. EC2 case sort decides the kind of virtual server used to dispatch the Amazon EMR center hubs. The case sort picked decides the virtual figuring environment for the hub: handling power, stockpiling limit, memory, et cetera.

The default center occasion sort for Hadoop 2.x is m1.large. Make certain to transform this to m3.xlarge. The m1.large occasion sort is not accessible
in each area. The m3.xlarge occurrence sort is appropriate for testing, developing, and light workloads. As a matter of course, Count is set to 2 for the center hubs. Demand spot is unchecked as a matter of course. This alternative determines whether to run center hubs on Spot Instances.

(e) Task: The errand occurrences run Hadoop assignments. Assignment occurrences don’t store information utilizing HDFS. At the point when errand hubs are utilized, they are contained in an undertaking occurrence bunch. EC2 example sort decides the kind of virtual server used to dispatch the Amazon EMR undertaking hubs. The case sort picked decides the virtual figuring environment for the hub: preparing power, stockpiling limit, memory, et cetera.

The default assignment occasion sort for Hadoop 2.x is m1.medium. This occasion sort is appropriate for testing, advancement, what’s more, light workloads. As a matter of course, Count is set to 0 for the assignment hubs. Utilizing errand hubs with Amazon EMR is discretionary. At the point when the occasion mean the errand hubs is 0, an assignment example gathering is not made. Demand spot is unchecked by default. This choice determines whether to run undertaking hubs on Spot Instances.

7. In the Security and Access area, for EC2 key pair, pick the key pair from the rundown and acknowledge the rest of the default choices.

(a) EC2 key pair: By default, the key pair alternative is set to Proceed without an EC2 key pair. This alternative keeps from utilizing SSH to interface with the expert, center, and assignment hubs. Amazon EC2 key pair is ought to be picked from the rundown.

(b) IAM client get to: All other IAM clients is chosen as a matter of course. This choice makes the group unmistakable and available to all IAM clients.
on the AWS account. If no other IAM clients have been picked, access to the group is limited to the current IAM client.

(c) IAM parts Default is chosen consequently. This alternative creates the default EMR part and default EC2 example profile. The EMR part and EC2 occasion profile are required while making a bunch utilizing the console. If Custom is selected, specific EMR part and EC2 occurrence profile should be indicated.

8. In the Bootstrap Actions section, accept the default option (none). Bootstrap actions are scripts that are executed during setup before Hadoop starts on every cluster node. They can be used to install additional software and customize the applications.

9. In the Steps section, accept the default options. These options are defined in the following table.

(a) Add step: By default, no user-defined steps are configured. A step is a unit of work submitted to the cluster. A step might contain one or more Hadoop jobs, or contain instructions to install or configure an application.

(b) Auto-terminate: By default, auto-terminate is set to No. This keeps the cluster running until it is terminated.

When it is set to yes, the cluster is automatically terminated after the last step is completed.

10. Choose Create cluster: On the Cluster Details page, proceed to the next step to run the Hive script as a cluster step and to use the Hue web interface to query the data.
4.4.3 Processing data with a Custom JAR using Hadoop in AWS

This section covers the basics of submitting a custom JAR step in Amazon EMR. Submitting a custom JAR step enables writing a script to process the data using the Java programming language.

To submit a custom JAR step using the console

1. Open the Amazon EMR console at https://console.aws.amazon.com/elasticmapreduce/.

2. In the Cluster List, select the name of cluster.

3. Scroll to the Steps section and expand it, then choose Add step.

4. In the Add Step dialog:

   (a) For Step type, choose Custom JAR.

   (b) For Name, accept the default name (Custom JAR) or type a new name.

   (c) For JAR S3 location, type or browse to the location of the JAR file. The value must be in the form s3://BucketName/path/JARfile.

   (d) For Arguments, type any required arguments as space-separated strings or leave the field blank.

   (e) For Action on failure, accept the default option (Continue).

5. Choose Add. The step appears in the console with a status of Pending.

6. The status of the step changes from Pending to Running to Completed as the step runs. To update the status, choose the Refresh icon above the Actions column.
Chapter 5

Evaluation

In this chapter, two datasets, Twitter and DBLP, are processed through ASJ and compared with SSS for better evaluation and the results are provided.

5.1 Twitter dataset

Here, experiments are conducted using Amazon Web Services (AWS) [71], a secure cloud services platform that offers large storage and computational capacity. EMR is the service functionality that is used to process big data. The observations are made using EMR clusters, with one master node, nine core nodes, and nine task nodes. Each node is an EC2 instance of M3 family. M3 instances provide a good balance of computation, memory and network resources. An M3 instance features high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) processors and SSD-based instance storage for fast I/O performance. Here, M3 instances of extra-large size are used in clusters with 15GB of memory each. Both algorithms have been run with varying thresholds of 0.7, 0.8 and 0.9 for better comparison. The dataset used here is a 60GB raw dataset of Twitter data in JSON format.

The data is first pre-processed to remove all stop words. Though there are other algorithms (like V-SMART) that implemented filtering techniques, SSS has been shown to outperform them in the literature on this dataset [69]. Hence, ASJ is compared
with SSS as a reference point.

Table 5.1: Execution time in seconds taken by ASJ (± standard error over 5 runs) and SSS for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
<th>SSS</th>
<th>Avg. Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>6,251 ± 122</td>
<td>10,084</td>
<td>38.0</td>
</tr>
<tr>
<td>0.8</td>
<td>4,586 ± 63</td>
<td>7,066</td>
<td>35.1</td>
</tr>
<tr>
<td>0.9</td>
<td>2,775 ± 35</td>
<td>3,732</td>
<td>25.6</td>
</tr>
</tbody>
</table>

Execution times for ASJ and SSS algorithms are given in Table 5.1. The average improvement is calculated using

\[
\frac{\text{SSS time} - \text{Avg. ASJ time}}{\text{SSS time}} \times 100\%.
\]  

(5.1)

There is a noticeable difference in the time taken for different thresholds. The time taken significantly reduces with increasing threshold since numerous candidate pairs are filtered in the early stages of the ASJ algorithm. The reduced computation time of ASJ compared to SSS is primarily due to the employment of suffix filtering in addition to the other filtering methods because filtering methods prune out the unique pairs in each stage before actually comparing the remaining pairs.

Table 5.2: Physical Memory Used (GB) by ASJ and SSS for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
<th>SSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>13.54</td>
<td>3,208.83</td>
</tr>
<tr>
<td>0.8</td>
<td>11.67</td>
<td>2,203.03</td>
</tr>
<tr>
<td>0.9</td>
<td>11.32</td>
<td>1,092.95</td>
</tr>
</tbody>
</table>

When the physical memory that used to process the dataset is compared between the two algorithms, ASJ uses much less memory (13GB) for processing the data. SSS takes up 3,208GB physical memory which is more than 200x that of ASJ. The
reason is that SSS carries the entire multiset file through all stages which takes greater memory to transfer while ASJ does not.

The actual time taken by the total map tasks and reduce tasks in stage-I and stage-II for ASJ are given in Table 5.3. It can be noted that the map phase in stage-II takes a long time, but the reduce phase finishes quickly. This is attributed to the positional and suffix filtering techniques in stage-II.

Table 5.3: Time (sec) taken by Map and Reduce tasks in ASJ to complete stage I and stage II for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Phases</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phases</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>978</td>
<td>562</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>Reduce</td>
<td>1,255</td>
<td>797</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>Stage II</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>5,121</td>
<td>3,344</td>
<td>1,681</td>
<td></td>
</tr>
<tr>
<td>Reduce</td>
<td>453</td>
<td>294</td>
<td>146</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Output generated (GB) by ASJ and SSS for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
<th>SSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>0.34</td>
<td>168.39</td>
</tr>
<tr>
<td>0.8</td>
<td>0.23</td>
<td>111.50</td>
</tr>
<tr>
<td>0.9</td>
<td>0.12</td>
<td>52.41</td>
</tr>
</tbody>
</table>

Considering output, the number of gigabytes written to file are represented in Table 5.4. SSS produces an output of 168.39 gigabytes while ASJ produces a much smaller output of 0.34GB. The low output size shows that ASJ provides good I/O and computational efficiency, and the massive output created points out that the SSS pruning techniques are not as effective.
Figure 5-1: Total output records produced by ASJ for Twitter dataset in all stages.

Table 5.5: Final stage output records produced by ASJ for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>2,030,998</td>
</tr>
<tr>
<td>0.8</td>
<td>1,154,240</td>
</tr>
<tr>
<td>0.9</td>
<td>518,263</td>
</tr>
</tbody>
</table>

The number of output records obtained from ASJ, after the final stage of filtering, are exhibited in Table 5.5. A comparison of output records produced by ASJ and SSS in Stage II is given in Fig. 5-1. The figure is drawn in logarithmic scale to compare. There is a significant difference in the number of output records written to file for each threshold. The output records of ASJ in 0.8 are 1.5x less than those in 0.7 while the output records in 0.9 are 2x less than that of 0.8. A graph has been plotted for the number of output records produced by ASJ in all stages for varying similarities is shown in Fig. 5-2.

The size of candidate pairs generated during each stage while processing the Twitter data are shown in Table 5.12. It can be observed that there is a compelling differ-
Figure 5-2: Number of output records produced by ASJ in all stages.

Table 5.6: Candidate pairs generated (MB) by ASJ in different stages

<table>
<thead>
<tr>
<th>Stages/Threshold</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage I</td>
<td>200,000</td>
<td>130,000</td>
<td>64,000</td>
</tr>
<tr>
<td>Stage II</td>
<td>322</td>
<td>207</td>
<td>103</td>
</tr>
<tr>
<td>Stage III</td>
<td>52</td>
<td>29</td>
<td>12</td>
</tr>
</tbody>
</table>

ence in the sizes of candidate pairs produced in various stages. It is evident that by implementing the suffix filtering technique, a significant number of candidate pairs are reduced in Stage II. Thus, by implementing suffix filtering in the same stage as positional filtering, many candidate pairs are pruned instead of being transferred to the next stage, therefore conserving memory. A notable difference can be observed between each of these stages. Although the prefix filtering technique is implemented, it is observed that prefix filtering had no effect on this algorithm because of how the data is preprocessed. However, an inverted index is implemented to generate candidate pairs in the first stage map phase.
5.2 DBLP dataset

DBLP [72] is a computer science bibliography website with approximately 3M publications. The original XML file was preprocessed to give a string concatenated with author’s names and title of publication. But some of the records did not have first author names and for these records, only published titles are processed. A unique integer is also added to this string for reference. It is processed on a single cluster local host Oracle VM Virtual Box with Ubuntu as the Operating System with 5GB memory. Hadoop 2.6.0 with Eclipse Juno is used in Ubuntu. Results have been noted for various thresholds of 0.6, 0.7, 0.8, and 0.9.

Table 5.7: Execution time in seconds taken by ASJ (± standard error over 5 runs) and SSS for DBLP dataset for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
<th>SSS</th>
<th>Avg. Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>61 ± 2</td>
<td>68</td>
<td>10.2</td>
</tr>
<tr>
<td>0.8</td>
<td>60 ± 2</td>
<td>66</td>
<td>9.09</td>
</tr>
<tr>
<td>0.9</td>
<td>56 ± 1</td>
<td>61</td>
<td>8.19</td>
</tr>
</tbody>
</table>

It can be observed from Table 5.9 that for DBLP dataset, the time required to run the dataset decreases. Decrease in the run time demonstrates efficiency and thus, less use of resources. There is roughly 8% to 10% increase in the execution times of DBLP datasets for ASJ in comparison to SSS algorithm. It takes approximately a minute to process the dataset for different thresholds. However, the difference of execution times by ASJ and SSS is not as significant. The execution times in Table 5.9 are taken over five runs and the average improvement is given by the Equation 5.1.

It is apparent from Table 5.8 that the output records generated for various thresholds do not have much difference between each other. That is because there are records that match more than 0.9 are more and it is found that most of them that are greater than 0.9 similarity match completely. These are duplicated multiple entries of the
Table 5.8: Final stage output records produced by ASJ for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>131</td>
</tr>
<tr>
<td>0.8</td>
<td>124</td>
</tr>
<tr>
<td>0.9</td>
<td>119</td>
</tr>
</tbody>
</table>

same record and these are joined as a single record in data cleaning.

Figure 5-3: Comparison of output produced by ASJ and SSS for DBLP dataset in all stages (KB).

When the output generated by ASJ and SSS is concerned, ASJ still does better in DBLP dataset but with reference to the Twitter dataset, the variation is not enormous. Since there was difference in inputs, it was obvious that the output produced in Twitter was in gigabytes (GB) and there is approximately 99% improvement but the output produced here is only in kilobytes (KB).

It can be noticed from Fig 5-4 that stage II does not have as much impact on DBLP dataset as Twitter since this is not a huge dataset and the strings are small comparatively. If the dataset is too small, adding suffix and positional filtering could
only be taking more time than just prefix and size filtering techniques because it would just be adding an extra stage for the same results where it just extends the length of the event. Hence, it can be deduced that suffix filtering and positional filtering does not have much effect here.

Table 5.9: Execution time in seconds taken by ASJ (± standard error over 5 runs) for DBLP dataset for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>66 ± 2</td>
</tr>
<tr>
<td>0.6</td>
<td>66 ± 1</td>
</tr>
<tr>
<td>0.65</td>
<td>63 ± 2</td>
</tr>
</tbody>
</table>

From Table 5.10, it can be learned that the records that have similarity less than 0.5 do not exist. It is obvious from Table 5.10 that for the similarity threshold 0.55, the output records are the same as 0.5.

It can be spotted from the Fig 5-5 that stage I (blue region) produces more output than the other two stages since it has only prefix and size filtering techniques which does not prune away every unique pair, as expected. While stage II reduces the pairs
Table 5.10: Final stage output records produced by ASJ for varying thresholds

<table>
<thead>
<tr>
<th>Threshold</th>
<th>ASJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>139</td>
</tr>
<tr>
<td>0.6</td>
<td>139</td>
</tr>
<tr>
<td>0.65</td>
<td>134</td>
</tr>
</tbody>
</table>

Table 5.11: Output bytes generated by ASJ in all stages

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Bytes generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>3449</td>
</tr>
<tr>
<td>0.7</td>
<td>3345</td>
</tr>
<tr>
<td>0.8</td>
<td>3283</td>
</tr>
<tr>
<td>0.9</td>
<td>3003</td>
</tr>
</tbody>
</table>

received from stage I and finally output records/pairs are obtained by calculating similarity in stage III.

One of the outputs generated is shown in Figure 5-6. Here, it is shown the first output record and fifth output record that have 1.0 and 0.75 similarity respectively. From Fig. 5-7 & 5-8 it can be observed that the record numbers 12535 and 13171 have the same title and hence, the similarity is 1.0. In Fig. 5-9 & 5-10, similarity of 0.75 in records 5600 and 16125 is shown.
Table 5.12: Size of candidate pairs generated (KB) by ASJ in different stages

<table>
<thead>
<tr>
<th>Stages/Threshold</th>
<th>0.65</th>
<th>0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage I</td>
<td>16.5</td>
<td>16.9</td>
</tr>
<tr>
<td>Stage II</td>
<td>9.6</td>
<td>9.8</td>
</tr>
<tr>
<td>Stage III</td>
<td>3.2</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Figure 5-5: Comparison of candidate pairs (KB) generated produced by ASJ for DBLP dataset in all stages.

Figure 5-6: One of the outputs produced by ASJ for DBLP dataset.
Figure 5-7: DBLP dataset for first output record.

Figure 5-8: DBLP dataset for first output record.

Figure 5-9: DBLP dataset for fifth output record.
20111125,16120,Shape 2009
20111125,16121,Imprecise Localization 2009
20111125,16122,Speech Analysis 2009
20111125,16123,Punctum Lacrimale 2009
20160215,16124,Speech Analysis 2015
20111125,16125,User Centered Design 2009
20111125,16126,Eye Centers 2009
20111125,16127,Internal Identification 2009
20111125,16128,Peg 2009
20111125,16129,Interface 2009
20160215 16130 Odor Biometrics 2015

Figure 5-10: DBLP dataset for fifth output record.
Chapter 6

Conclusion

This study has proposed and evaluated a three-stage similarity join algorithm in MapReduce, called ASJ, that improves and advances prefix, size, positional and suffix filtering techniques as currently integrated with multiset similarity joins on Twitter and DBLP datasets and the work was compared to SSS algorithm in detail.

In stage I map phase, an inverted index is applied so that the elements sharing the same prefix and data element can be grouped together and candidate pairs are generated in the reduce phase. To minimize the huge number of candidate pairs created, size filtering is applied in the reduce phase. By applying size filtering in stage I itself, as soon as candidate pairs as generated, unnecessary transfer of data to stage II is eliminated.

Suffix filtering and positional filtering techniques are extended to multisets and are applied together in stage II reduce phase. By applying these techniques together, immediate emission of unique pairs without carrying over is done. Multiset file used by SSS to transfer data in every phase and stage is not a feasible option, since it acquires abundant memory. That is eliminated in ASJ. It is shown that by implementing suffix filtering in addition to other filtering techniques, a significant number of candidate pairs produced in the course of the algorithm are reduced, thus improving I/O, network and computational efficiency. The time taken to reload the data is
eliminated by performing suffix filtering in Stage II. Hence memory, space and time are conserved. ASJ shows 8% to 40% increase in execution time and 99% memory efficient.

It should also be noted that for smaller datasets like DBLP, though ASJ demonstrates better results than SSS in every aspect, it takes time to pass through the stage II. It would be safe to say ASJ will have a significant impact on large datasets.

Overall, this thesis shows that ASJ has exceptional pruning results and that it exceeds the current state of the art, the SSS algorithm, in terms of memory usage and computation time required.

As future work, this study can be extended by focusing on experimenting with larger datasets greater than 100 GB. Further, the algorithm can be improved by combining the second and third stages. Thus performing positional and suffix techniques in stage II map phase and carrying out similarity calculations in reduce phase. This could mean performing it in two stages, and therefore providing I/O and time efficiency.
References


[44] C. Li, J. Lu, and Y. Lu, “Efficient merging and filtering algorithms for approx-


Appendix A

Source code for Stage I

The source code for Adept Similarity Joins algorithm is also available at [Github](https://github.com).

A.1 Map phase

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.StringTokenizer;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class OnlMap2 extends MapReduceBase implements Mapper<LongWritable,
Text, Text, Text>
{
```

public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {

    StringTokenizer st = new StringTokenizer(value.toString());
    String t = st.nextToken(); //Token
    String uid = st.nextToken(); //User ID
    String uid_size = st.nextToken(); //Size of that user ID
    String frequency = st.nextToken(); //frequency of that user ID
    String data = st.nextToken(); //Data in the UID
    String str_key = t + " " + data;
    String str_val = uid + " " + uid_size + " " + frequency;
    //System.out.println("frequency:" + uid_size);
    output.collect(new Text(str_key), new Text(str_val));
}

A.2 Reduce Phase

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.StringTokenizer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class OnlReduce2StopWords extends MapReduceBase implements 
Reducer<Text, Text, Text, Text> {

    public void reduce (Text key, Iterator<Text> values, OutputCollector 
<Text, Text> output, Reporter reporter) throws IOException, 
NumberFormatException {

        int count = 0;
        int sum = 0;

        ArrayList<CustomMapGI> alcmap = new ArrayList<CustomMapGI>();
        StringTokenizer st = new StringTokenizer(key.toString());
        String t = st.nextToken();
        String data = st.nextToken();
        while(values.hasNext())
        {
            String val = values.next().toString();
            StringTokenizer stv = new StringTokenizer(val);
            String uid = stv.nextToken();
            String uid_size = stv.nextToken();
            System.out.println("uid_size:" +uid_size);
            String frequency = stv.nextToken();
            CustomMapGI cmap = new CustomMapGI(uid, uid_size, frequency);

            // Add code here for processing
        }
    }
}
alcmap.add(cmap);
sum += Integer.parseInt(frequency);
count++;
}

// System.out.println("alcamp size:" +alcmap.size());
if(alcmap.size()<5000)
{
    for(int i = 0; i<alcmap.size(); i++)
    {
        for(int j = 1; j<alcmap.size(); j++)
        {
            if((j>i)&&(!alcmap.get(i).equals(alcmap.get(j))))
            {
                String uidi = alcmap.get(i).getUid();
                String uidj = alcmap.get(j).getUid();
                ArrayList<String> alsort = new ArrayList<String>();
                alsort.add(uidi);
                alsort.add(uidj);
                String i_size = null;
                String j_size = null;
                String freq_i = null;
                String freq_j = null;
                if(alcmap.get(i).getUid().equals(alsort.get(0))
                {
                    i_size = alcmap.get(i).getSize();
                    j_size = alcmap.get(j).getSize();
                }
freq_i = alcmap.get(i).getFrequency();
freq_j = alcmap.get(j).getFrequency();
}
Collections.sort(alsort);
int f_i = Integer.parseInt(freq_i);
int f_j = Integer.parseInt(freq_j);
int si = Integer.parseInt(i_size);
int sj = Integer.parseInt(j_size);
int small = 0;
int big = 0;
if(si<sj)
{
    small = si;
    big = sj;
}
else
{
    small = sj;
    big = si;
}
double small1 = (double) small;
double big1 = (double)big;
double filter = small1/big1;
System.out.println("filter:" +filter);
double checker = 0.7;
if(filter > checker)
{

String mul_key = t + "+uidi" +uidj" +i_size" +j_size;
String str_val = f_i" +f_j" +data;

output.collect(new Text(mul_key),
new Text(str_val));
Appendix B

Source code for Stage II

B.1 Map phase

```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class OnlMap3 extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {

    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
    }

    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
    }
}
```
String t = tokenizer.nextToken();
String uidi = tokenizer.nextToken();
String uidj = tokenizer.nextToken();
String i_size = tokenizer.nextToken();
String j_size = tokenizer.nextToken();
String freq_i = tokenizer.nextToken();
String freq_j = tokenizer.nextToken();
String data = tokenizer.nextToken();
String str_key = t + " " + uidi + " " + uidj;
String str_val = i_size + " " + j_size + " " + freq_i + " " + freq_j + " " + data;
output.collect(new Text(str_key), new Text(str_val));
}

}

B.2 Reduce phase

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.StringTokenizer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

80
public class OnlReduce2StopWords extends MapReduceBase implements 
Reducer<Text, Text, Text, Text>{

public void reduce (Text key, Iterator<Text> values, OutputCollector <Text, Text> output, Reporter reporter) throws IOException, 
NumberFormatException {

    int p = 0; % alpha p is p
    int q = 0;
    boolean passpf=false;
    boolean passfc=false;
    boolean passsf = false;
    double z = 0;
    int sum = 0;
    int count = 0;
    StringTokenizer st = new StringTokenizer(key.toString());
    String t= st.nextToken();
    String uuid = st.nextToken();
    String uidj = st.nextToken();

    int fi = 0;
    int fj = 0;
    int s_i = 0;
    int s_j = 0;
    String data= "";
    String i_size = "";
}
String j_size = "";

while(values.hasNext())
{
    String val = values.next().toString();
    StringTokenizer stv = new StringTokenizer(val);
    i_size = stv.nextToken();
    j_size = stv.nextToken();
    String freq_i = stv.nextToken();
    String freq_j = stv.nextToken();
    data = stv.nextToken();
    s_i = Integer.parseInt(i_size);
    s_j = Integer.parseInt(j_size);
    fi = Integer.parseInt(freq_i);
    fj = Integer.parseInt(freq_j);
}

double sii = (double)s_i;
double sjj = (double)s_j;
int intersection = 0;
int maxi = 0;
if (fi < fj)
{
    q = fi;
    intersection = fi;
    maxi = fj;
}
else
{
    q = fj;
    intersection = fj;
    maxi = fi;
}

p += q; // alpha p is p
double alphap = (double)p;

z = (7/17)*(sii+sjj); //z is alpha from stage -1 map phase
int pos_i=0;
int pos_j=0;
if (!values.hasNext())
{
    pos_i += fi;
    pos_j += fj;
    int part_i = pos_i+fi;
    int part_j = pos_j+fj;
    int mini= s_i-part_i;
    int minj = s_j-part_j;
    double min_i = (double)mini;
    double min_j= (double)minj;

    if (mini<minj)
    {
        double overlap = min_i;
        double ubound = overlap + alphap;

if (ubound >= z)
{
    passpf=true;
    double suf_mx = (((0.7)*s_i)-1);
    double suf_my = (((0.7)*s_j)-1);
    int l_x = 0;
    l_x += fi;
    double r_x = suf_mx - (l_x+fi);
    int l_y = 0;
    l_y += fj;
    double r_y = suf_my - (l_y+fj);
    int diff = 0;

    if (pos_i == pos_j)
    {
        diff = q;
    }
    else
    {
        diff = fi + fj;
    }

    double hs = Math.abs(l_x-l_y) + Math.abs(r_x-r_y) + diff;
    double hpmin = (pos_i+fi) + (pos_j+fj) - 2*p;
    double hmax = s_i + s_j - (2*z) - hpmin;
    if (hs<hmax)
    {
        // code here
    }
passsf = true;

String str_key = t + "+" + uid_i + "+" + uid_j;

String str_val = s_i + "+" + s_j + "+" + intersection + "+" + maxi + "+" + data;

output.collect(new Text(str_key), new Text(str_val));
Appendix C

Source code for stage III

C.1 Map phase

```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class OnlMap4 extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
    }

    public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
    }
```

86
C.2 Reduce phase

```java
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
```

```java
String t = tokenizer.nextToken();
String uidi = tokenizer.nextToken();
String uidj = tokenizer.nextToken();
String i_size = tokenizer.nextToken();
String j_size = tokenizer.nextToken();
String intersection = tokenizer.nextToken();
String maxi = tokenizer.nextToken();
String data = tokenizer.nextToken();
String str_key = t + " " + uidi + " " + uidj + " " + i_size + " " + j_size;
String str_val = intersection + " " + maxi;
output.collect(new Text(str_key), new Text(str_val));
```
public class OnlReduce4 extends MapReduceBase implements Reducer
<Text, Text, Text, Text>{

public void reduce (Text key, Iterator<Text> values, OutputCollector <Text, Text> output, Reporter reporter) throws IOException {

    StringTokenizer st = new StringTokenizer(key.toString());
    String t = st.nextToken();
    String uidi = st.nextToken();
    String uidj = st.nextToken();
    String i_size = st.nextToken();
    String j_size = st.nextToken();
    String j_size = st.nextToken();
    int inter=0;
    int max = 0;
    int interr=0;
    int maxx=0;

    while(values.hasNext())
    {
        String val = values.next().toString();
        StringTokenizer stv = new StringTokenizer(val);
        String intersection = stv.nextToken();
        String maxi = stv.nextToken();
        inter = Integer.parseInt(intersection);
        max = Integer.parseInt(maxi);
        interr = interr+inter;

    }

}
maxx = maxx+max;
}

System.out.println("inter: "+inter);
System.out.println("max: "+max);
double interrr = (double) interr;
double maxxx = (double)maxx;
double similarity = ((interrr)/(maxxx));

if(similarity>0.7)
{
    String str_key = t+" "+uidi+" "+uidj;
    String str_val = String.valueOf(similarity);
    output.collect(new Text(str_key), new Text(str_val));
}
}