View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by SJSU ScholarWorks

San Jose State University
SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 5-26-2016

EFFICIENT PAIR-WISE SIMILARITY COMPUTATION USING APACHE SPARK

Parineetha Gandhi Tirumali
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Databases and Information Systems Commons

Recommended Citation

Tirumali, Parineetha Gandhi, "EFFICIENT PAIR-WISE SIMILARITY COMPUTATION USING APACHE SPARK" (2016). *Master's Projects*. 479.
DOI: https://doi.org/10.31979/etd.sh8a-3gyv
https://scholarworks.sjsu.edu/etd_projects/479

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

EFFICIENT PAIR-WISE SIMILARITY COMPUTATION USING APACHE SPARK

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

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

By
Parineetha Gandhi Tirumali
May 2016

©2016
Parineetha Gandhi Tirumali
ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

EFFICIENT PAIR-WISE SIMILARITY COMPUTATION USING APACHE SPARK

by
Parineetha Gandhi Tirumali

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSE STATE UNIVERSITY
May 2016

Dr. Tran Duc Thanh Department of Computer Science.
Dr. Thomas Austin Department of Computer Science.
Mr. Subrahmanyam Bolla Manager, VMWare.

ABSTRACT

EFFICIENT PAIR-WISE SIMILARITY COMPUTATION USING APACHE SPARK

by Parineetha Gandhi Tirumali

Entity matching is the process of identifying different manifestations of the same real world entity. These entities can be referred to as objects(string) or data instances. These entities are in turn split over several databases or clusters based on the signatures of the entities. When entity matching algorithms are performed on these databases or clusters, there is a high possibility that a particular entity pair is compared more than once. The number of comparison for any two entities depend on the number of common signatures or keys they possess. This effects the performance of any entity matching algorithm. This paper is the implementation of the algorithm written by Erhard Rahm et al. for performing redundancy free pair-wise similarity computation using MapReduce. As an improvisation to the existing implementation, this project aims to implement the algorithm in Apache Spark in standalone mode for sample of data and in cluster mode for large volume of data.

ACKNOWLEDGEMENTS

I would like to thank the following people and many others who aren't named here.

I would like to express my gratitude to Professor Tran for his continuous support and guidance throughout the completion of this project.

I would like to thank my committee members Dr. Thomas Austin and Mr. Subrahmanyam Bolla for their contribution.

I would like to thank my parents, family and my husband for their cooperation.

Table of Contents

ABSTRACT iv

ACKNOWLEDGEMENTS v

List of Figures viii

List of Tables x

1. Introduction 1

2. Related Works 3

  2.1 Terminology 3

    2.1.1 Entity 3

    2.1.2 Signature 3

    2.1.3 Matching 3

  2.2 Pair-wise similarity computation(PSC) 3

  2.3 Spark Implementation of Map Reduce 3

3. Problem Definition 6

  3.1 Challenges 7

  3.2 Signature Function 7

  3.3 Map Function 12

  3.4 Reducer Function 12

  3.5 Real Time Example 14

4. Implementation Details 17

  4.1 Baseline Implementation 1 17

  4.2 Baseline Implementation 2 21

    4.2.1 Hive Context 21

    4.2.2 Data Frames 22

4.2.3 Window Concept in Spark 23

5. Performance Evaluation 26

5.1 Apache Spark Cluster Setup 27

5.2 Launching and testing the cluster 33

6. Conclusion 35

7. References 36

List of Figures

Figure 1: Referencing same paper object multiple times 1
Figure 2: Multiple entries for the same product 1
Figure 3: Spark Execution Framework 4
Figure 4: Python Spark Data Flow Architecture 5
Figure 5: Worker Node Data Flow 5
Figure 6: Map Reduce Data Flow(1) 6
Figure 7: Difference between Map Reduce and Spark 7
Figure 8: Example of Pair-wise similarity computation 8
Figure 9: Signatures for two pass blocking 9
Figure 10: Output of Map phase 10
Figure 11: Output of the reduce phase 11
Figure 12: Map Reduce phase after improvising the algorithm 13
Figure 13: Sample Table 14
Figure 14: Key value pairs generated by Map 14
Figure 15: Grouping similar entities 15
Figure 16: Sorting the keys associated with each entity 15
Figure 17: Entity pairs 16
Figure 18: Snapshot of indexed file 18
Figure 19: Code snippet to generate key value pairs from map 18
Figure 20: Snapshot of indexed file contents 19
Figure 21: Snapshot of Entity, SortedSignatures 19
Figure 22: Code snippet for generating entity pairs 20
Figure 23: Snapshot of Key value pairs in the form of key and pairs of entities 20
Figure 24: Generation of Entity Pairs 20
Figure 25: List of attributes and objects 23
Figure 26: List of attributes, objects and previous values 24
Figure 27: List of attributes and object, previous pairs 24
Figure 28: Algorithm for Map Phase 25

Figure 29:Algorithm for Reduce Phase 25
Figure 30:Algorithm for Overlap 25
Figure 31: Execution times comparison 27
Figure 32: Screenshot of two VMs in VMWare WorkStation 27
Figure 33: Screenshot showing Java version on VM 28
Figure 34: Screenshot for Scala download 28
Figure 35: Screenshot showing Scala version on VM 28
Figure 36: Screenshot of Master Node 29
Figure 37: Screenshot of Worker Node 29
Figure 38: Screenshot for installing ssh 30
Figure 39: Screenshot showing Master connection 30
Figure 40: Spark version 31
Figure 41: Files in conf folder 31
Figure 42: Updated Slaves file 32
Figure 43: Updated spark-env.sh file 32
Figure 44: Files in sbin folder 34

List of Tables

Table 1: Execution time with and without redundant pairs 26

1. Introduction

Entity matching is the process of identifying different manifestations of the same real world entity. An entity can be an object, data instance or a record. Examples of manifestations and objects include: different ways of addressing(names, email addresses) the same person; web pages with different descriptions of the same business; different photos of the same object and so on. The matching is performed by implementing several techniques like numerical matching approach, rule-based matching approach and workflow-based matching approach[1].

Some of the examples of entity matching are

Example 1:

Title	Author	Venue	Year
The merge/purge problem for large databases	M.A. Hernandez, S.J. Stolfo	Proceedings of the ACM SIGMOD international conference	
The Merge/Purge problem for Large Databases	A.H. Mauricio, J.S. Stolfo	Proc. of the 1995 ACM SIGMOD conference on management	1995
andez, SJ Stolfo The merge/purge problem for Large Databases	M. Hern	Proceedings of the 1995 ACM SIGMOD conference on management	1995

Figure 1: Referencing same paper object multiple times

Example 2:

Canon VIXIA HF S10 Camcorder - 1080p - 8.59 MP - 10 x optical zoom
Flash card, 32 GB, 1y warranty, F/1.8-3.0
The VIXIA HF S10 delivers brilliant video and photos through a Canon exclusive 8.59 megapixel CMOS image sensor and the latest version of Canon's advanced image processor, ...
12 reviews - Add to Shopping List
$975 new
from 52 sellers
Compare prices

Canon (VIXIA) HF S10 iVIS Dual Flash Memory Camcorder
Canon HF S10 iVIS Dual Flash Memory CamcorderSPECIAL SALE PRICE: $899 Display both English/Japanese + we supplu all English manuals in English as PDF. ...
Add to Shopping List
$899.00 new
Made in Japan Online

Canon VIXIA HF S10
Dual Flash Memory High Definition Camcorder The Next Step Forward in HD Video Canon has a well-known and highly-regarded reputation for optical excellence, ...
Add to Shopping List
$999.00 new
Performance Audio
2 seller ratings

Canon VIXIA HF S100 Flash Memory Camcorder
***Canon Video HF S100 Instant Rebate Receive $200 with your purchase of a new Canon VIXIA HF S100 Flash Memory Camcorder. (Price above includes $200 ...
Add to Shopping List
$899.95 new
Arlingtoncamera.com
5 seller ratings

Canon Vixia Hf S10 Care & Cleaning
Care & Cleaning Digital Camera/Camcorder Deluxe Cleaning Kit with LCD Screen Guard Canon VIXIA HF S10 Camcorders Care & Cleaning.
Add to Shopping List
$2.99 new
shop.com
38 seller ratings

Figure 2: Multiple entries for the same product

To say if two objects or two products or two people are same or not we need to compare them. To know if two entities are a match or non match, we need to first compare the pairs of entities. Due to the different manifestation of an entity and presence of multiple signatures or attributes for each entity, the entities will get compared more than once. The other reason for a particular pair of entity to be compared more than once is due to the presence of overlapping clusters. One of the processes for entity matching is blocking, this blocking is performed based on a blocking key. There is a possibility that an entity can have more than one blocking key, because of which the entities gets to share more than one cluster. So the comparison takes place more than once, due to which the efficiency deteriorates.

What motivates for Entity matching is linking census records, public health, web search, comparison shopping, portals integration from multiple sources, electronic marketplaces, integrating genomic data in medical genetics, monitoring events in the sky in the field of astrophysics.

The remaining part of this paper is organized as: section 2 gives a brief on related works done in pair wise similarity computation and its drawbacks. Section 3 describes about the problem definition and section 4 explains how this problem has been addressed.

Entities are distributed among the clusters or databases and when a comparison is performed on these clusters or databases, there is a high chance that the entities are compared redundantly. This reduces the efficiency of entity matching. One naive way of increasing the efficiency in terms of speed is using map reduce. But this does not solve the problem of redundant entity comparison completely. So this paper tries to solve the problem of redundant entity comparison using the concept of data frames and windows in Apache Spark.

The terms Entity and Object, and the terms Signature and Keys are one and the same throughout the paper.

2. Related Works

Here after in this paper I use the following terminology to explain the concepts. This chapter gives a brief about these terminologies

2.1 Terminology

2.1.1 Entity

An entity in real world can be a person, product or any object which has attributes associated to it. An entity can also be referred to as an object, string or a document. Example: name, product.

2.1.2 Signature

A signature is associated with the entity, it can be a token, blocking key or set of terms. Example: category.substr(0,3) or manufacturer.

2.1.3 Matching

It is a process of comparing two entities and saying if they are a match or non-match.

Examples: Matching products for comparison shopping. Finding duplicate entries of customers in enterprise database

2.2 Pair-wise similarity computation(PSC)

Pair-wise similarity computations is an important concept in data related applications like entity resolution, clustering based on entities, etc. Groups of entities with same signature fall into one cluster and this is called clustering. During this process there is a high chance that entities getting duplicated in case of common signatures. How to deal with such scenario is the whole idea of implementing this paper.

2.3 Spark Implementation of Map Reduce

Apache Spark is an open source project found by UC Berkeley AMP Labs, the main motive of this project was to use in-memory, distributed data structure to speed up data processing over Hadoop. Map reduce concept was the early trial of making the

process execution faster over distributed data structure, but introduction of spark made programs way more flexible and faster compared to map reduce alone.

In map reduce framework, few tasks are assigned to map and few to reducers. But, spark has a generic executor(JVM) depending on a situation executes map stages and reduces. JVM is core where all computation is executed, it is also an interface for other ecosystems like Hadoop. Consider we need to process 1TB of data on AWS, and the one worker node processes 1GB of data in map stage the result is stored as 1 RDD.

Using Java or Scala will run the process directly on JVM. But for python the execution framework is different, it has several python or pyspark processes, generally one per task depending on the application. These processes are connected to JVM and data is shipped from JVM to python for processing.

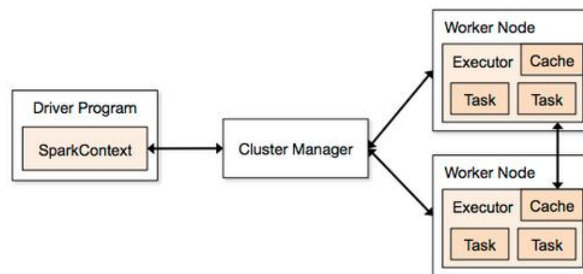


Figure 3: Spark Execution Framework

There can be several such worker nodes but there should be only one manager to provision or restart workers. This is called Cluster Manager. The object that connects and holds the cluster in spark is spark context. Spark's driver program directs the operations by initializing the spark context. Spark's actions and transformations are initialized in this spark context and when the program gets executed the worker nodes kick starts and process the data. Following figures show how the data flow when using python in Spark.

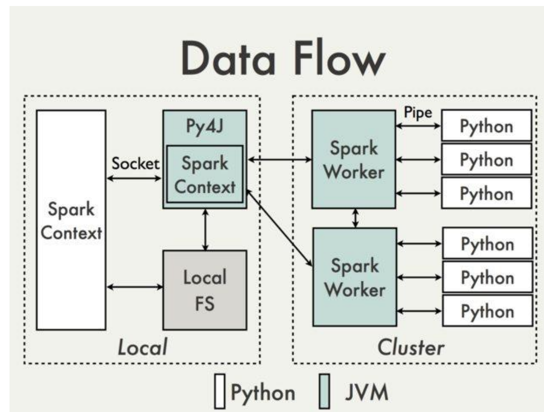


Figure 4: Python Spark Data Flow Architecture

Spark supports two interfaces of cluster management: yarn and standalone. Yarn is Hadoop's cluster manager which can be used with Hadoop map reduce and spark. Whereas a standalone interface has special spark process which takes care of starting the nodes that are failing.

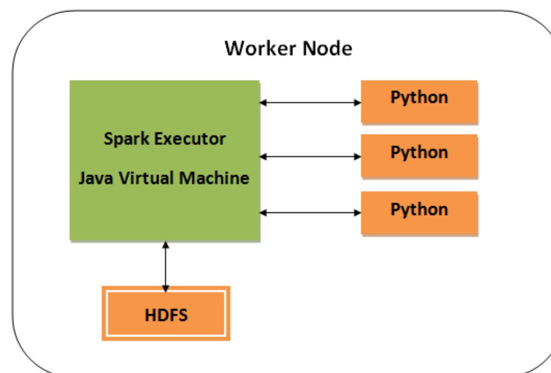


Figure 5: Worker Node Data Flow

3. Problem Definition

Data quality play an important role for entity matching. Big data is massive representation of data and to find matching entities is very crucial and challenging task. This project mainly implements the concept of pair wise redundancy free comparison using Apache Spark. The basic approach for pair-wise similarity computation takes Cartesian product of the entity pairs. Cartesian product gives a complexity of $O(n^2)$ which is very high in terms of big data. This can be improvised by using Map Reduce concept to parallelize the computation which in turn speeds up the process but the quadratic complexity seems to be almost same even after using Map Reduce. So the paper referred modifies the algorithm for reduce phase. Following figure shows how the basic map reduce works.

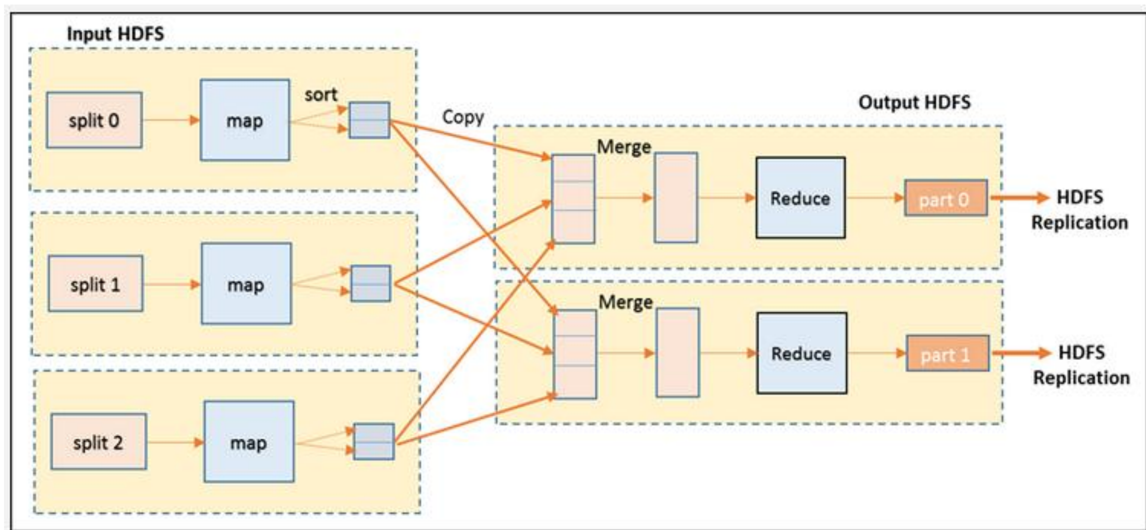


Figure 6: Map Reduce Data Flow(1)

Map Reduce alone sometimes is not so efficient when compared to Spark. Following are few differences between Map Reduce and Spark and we can clearly see that Spark out performs extraordinarily when compared to Map Reduce.

Criteria	Map Reduce	Spark
Conciseness	Plain MR has a lot of boiler plate	Almost no boilerplate
Performance	High latency	very fast compared to MR
Testability	Possible via libraries, but non trivial	Very much easy
Iterative processing	Non trivial	straight forward
Exploration of data	Not possible easily	Spark shell allows quick and easy data exploration
SQL like interface	Via Hive	Build in as SparkSQL
Fault Tolerance	Inheranlty able to handle fault tolerance via persisting the results of each of phases	Exploits immutability of RDD to enable fault tolerance
Eco system	lots of tools available but integration is not quite seamless, requiring lot of effort for their seamless integration	Unifies lot of interfaces like SQL, stream processing etc into single abstraction of RDD
In memory computations	not possible	possible

Figure 7: Difference between Map Reduce and Spark

3.1 Challenges

Matching entities while dealing with big data can be a tedious process and the quality of the data place a major role. When the data is so enormous in size it is obvious that it can be heterogeneous and there lies the challenging part. Heterogeneous data is unclean, unstructured and incomplete. With the growing data, applications and relationship between various sources of data, the need for matching is also growing. With this growth matching names with names is not as important as matching Amazon profiles with browsing history on Google and friends profile on Facebook. Larger datasets need efficient parallel techniques to process them.

3.2 Signature Function

The main problem in entity matching is that a particular entity pair comparison takes place many times, this leads to redundant pair comparison which reduces the efficiency of entity matching. The solution for elimination of redundant pair comparisons can be achieved by efficiently integrating with a parallel MR implementation.

Redundant comparison takes place when there are more than one common signatures between two entities. The basic map reduce can be improved by introducing the concept

of clustering. The search space to match particular entity is reduced by grouping them with the entities of similar entities, and this group forms a cluster. Every cluster has the entities which are similar and the comparison takes place for pairs of entities present within the cluster.

For every entity in a group of entities **O**, a sub group of attributes **s** are generated using the following signature function

$$\sigma: O \rightarrow \mathcal{P}(S)$$

This function takes group of entities and the attributes **S** as input and generate subset of attributes $s \subseteq S$ for each entity $o \in O$. The pair-wise similarity algorithm generates the similarity for all entity pairs that have minimum one common attribute.

$$\{(o1, o2) | (o1,o2) \in O \times O \wedge o1 \neq o2 \wedge \sigma(o1) \cap \sigma(o2) \neq \emptyset\}$$

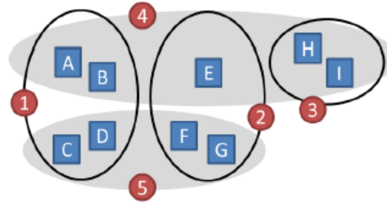


Figure 8: Example of Pair-wise similarity computation

In real time A, B, C.. can represents the product name, person name etc. 1, 2, 3.. can be price of a product, manufacturer or substring of title or category etc.

Consider A, B, C, D, E, F, G, H, I are few entities having 1, 2, 3, 4, 5 as keys or signatures. If these entities are not clustered there is a high possibility that these entities can be compared more than once. So the blocking algorithm is first performed to reduce the search space for matching. The figure shown above looks like it is the result of two pass blocking. Because of the presence of the more number of common signatures the entity pairs get compared once for each common signature. Generating signatures for the entities is part of the blocking phase. Blocking can be done based on one pass, or two

pass or multi pass. The result obtained with multi pass is considered to be more accurate to find duplicates than single pass, because the entities which are not grouped in the first block gets grouped properly in the subsequent block phase. As the number of passes increases the size of the cluster gets smaller. The drawback of smaller cluster size is the similar entities pairs are missed. These missed entity pairs will never get compared with each other. The concept of blocking is beyond the scope of this project. Deciding on how many keys to generate and what keys to generate for entities is a difficult task, which depends on number of entities per cluster. For example, if the employee entities are clustered based on the address, there is a possibility that the same employees might be placed in different clusters if the address is slightly varying or missing.

After performing the two pass blocking on Figure 8 the resultant signature function generates signatures as shown in Figure 8

Object	Signature
A	{1, 4}
B	{1, 4}
C	{1, 5}
D	{1, 5}
E	{2, 4}
F	{2, 5}
G	{2, 5}
H	{3, 4}
I	{3, 4}

Figure 9: Signatures for two pass blocking

The first blocking phase generates three clusters with keys 1, 2, 3 and the second blocking phase generates clusters with 4, 5 keys. The map phase generates key value pairs for each entity. The output of the map phase is fed to the reducers through the partitioners. The partitioner performs some function over the keys, in this particular example the function is finding the modulo. The key value is divided by the number of keys and as per the result the partitioner send the key value pairs to the reducers. The signatures 1, 3, 5 are passed to one reducer and the signatures with 2 and 4 are passed to the other reducer. The output of the map phase is shown below.

Map: Signatures

Obj
A
B
C
D
E

Key	Value
1	A
4	A
1	B
4	B
1	C
5	C
1	D
5	D
2	E
4	E

Obj
F
G
H
I

Key	Value
2	F
5	F
2	G
5	G
3	H
4	H
3	I
4	I

Figure 10: Output of Map phase

Pair-wise comparison takes place in reducer for each key. Due to the presence of the common signatures the entities are compared redundantly in the reduce phase. The entities that are compared redundantly are underlined in the figure shown below. This is due to the presence of overlapping cluster. So there is a need to change the processing of the reduce phase which can avoid redundant comparisons. The output of the reduce phase before changing the algorithm is shown in Figure 10.

Reduce: Pair Comparisons

Key	Value
1	A
1	B
1	C
1	D
3	H
3	I
5	C
5	D
5	F
5	G

Pairs
A-B, A-C, A-D, B-C, B-D, C-D
H-I
C-D, C-F, C-G, D-F, D-G, F-G

Key	Value
2	E
2	F
2	G
4	A
4	B
4	E
4	H
4	I

Pairs
E-F, E-G, F-G
A-B, A-E, A-H, A-I, B-E, B-H, B-I, E-H, E-I, H-I

Figure 11: Output of the reduce phase

The main reason for the redundant comparison to take place is that one reducer is not aware of what entity pairs other reducers are processing. So to make the reducers smarter we need to implement a small function to calculate the minimum value among the list of common keys for any two entity pairs.

$$l \in \min(\sigma(o1) \cap \sigma(o2))$$

There can be various other approaches to solve this problem instead of just finding the smallest key among the list of common keys. Now the reducer that handles the signature l is responsible for comparing the entities o1and o2. No other reducer will compare these two entities again. How does this work? The reducer receives entities o1 and o2 as input and the partitioner. The reducers checks if there is any signature less than the current signature produced by the partitioner. If there exists a signature less than that then the reducer will not compare these two entities as it assumes that this pair is taken care by the other reducer. If there is no signature less than the one produced by the partitioner then the reducer takes that entity pair and compares.

3.3 Map Function

The output of the map function is modified from just generating the key value pairs to generating the subgroup of keys smaller than the key received. Initially the map outputted the list of keys for a particular entity i.e.,

$$\boldsymbol{\sigma(o) = \{s_1, s_2, s_3,..s_n\}}$$

$$\text{Let } \boldsymbol{\sigma_{si}(o) = s \in \sigma(o) \mid s < s_i}$$

After improvisation the map function now emits

$$\boldsymbol{(\, s_i\, ,\, [\, o\, ,\, \sigma_{si}(o)])} \text{ for every } \boldsymbol{1 \leq i \leq n}$$

3.4 Reducer Function

Reducer takes the above input and for present key k and the entity pair similar to the one shown above performs an extra step of checking if the two entities have disjoint key set. For a given pair $\boldsymbol{(\, [\, o_1\, ,\, \sigma_k(o_1)\,],\, [\, o_2\, ,\, \sigma_k(o_2)\,]\,)}$ the reducer checks if

$$\boldsymbol{\sigma_k(o_1) \cap \sigma_k(o_2) = \emptyset}$$

If they are disjoint the reducer makes the current key as the least common key and the two entities o1 and o2 are compared. If these two sets are not mutually exclusive then it means that there is/are smaller keys k' is present for that pair of entity (o_1, o_2). So in this scenario the key k is not considered.

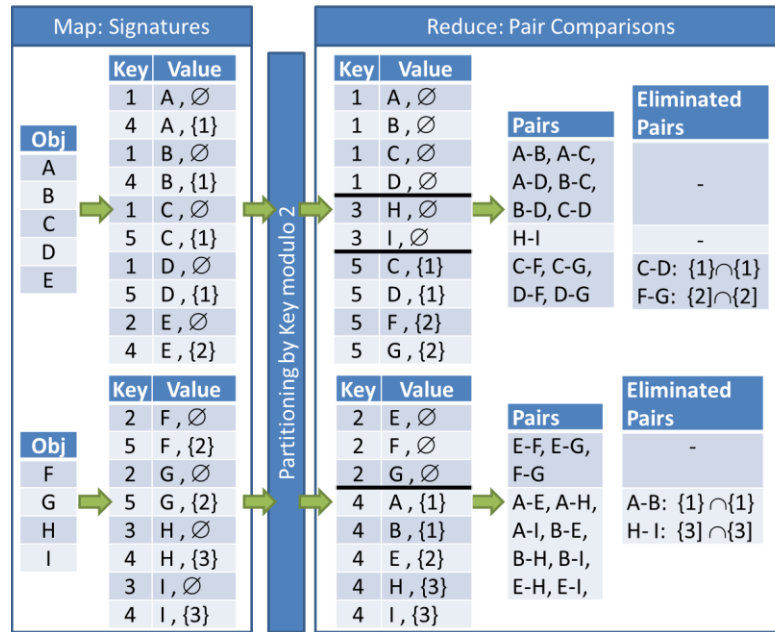


Figure 12: Map Reduce phase after improvising the algorithm

Now the map phase emits key value pairs along with the keys smaller than the current key. For example, consider the entity A which has two common keys 1,4 where **σ(A)={1, 4}**. The key value pair for the first time would be **(1, [A, Ø])**, where Ø represents that there are no keys smaller than the current key 1. Now when the key 4 is considered, the map function gives **(4, [A, {1}])**, which means that there is a key smaller than the current key 4. As **σ(B)={1, 4}** the pair A-B is compared with the key 1. Later when key 4 is considered then the **σ_4(A)= (4, [A, {1}])** has a subset of the key {1}.

The initial (k, v) i.e., **(1, [A, Ø])** which is passed to the first reducer is compared against other entities who has common signature 1. In this case it is B, C, D. So the pairs A-B, A-C, A-D, B-C, B-D, C-D are compared. Now the entities that share the signature 4 are sent to other reducer. As **σ(E)={2, 4}, σ(H)={3, 4}, σ(I)={3, 4}** and now the pair A-B is ignored. In total, this process generated 26 pairs before the improvement of the algorithm. After the improvements made to the reducer it eliminated 4 pairs which is 15% less than the actual. So there is performance improvement.

It is implemented in such a way that the entities with same keys fall under the same window and this window moves over rows of same obj and sorted by attributes. This approach uses two map jobs, one to emit key value pair for each key and another map job is used to group together the pairs having the same key.

3.5 Real Time Example

Consider that a dataset has following records

1	pari	sjsu	tirumali
2	pari	JNTU	tiru
3	pari	SJ	T
4	pari	CA	Gandhi
5	pari	CA	T.G
6	pari	SJ	T.G
7	pari	SJS	T
8	pari	USA	tirumali
9	pari	India	tirumali
10	pari	Earth	tiru
11	pari	pari	pari
12	tiru	pari	tiru

Figure 13: Sample Table

And consider that there is one pass blocking and the map phase generates following key value pairs

pari, 1	pari, 5	pari, 9
sjsu, 1	CA, 5	India, 9
tirumali, 1	T.G, 5	tirumali, 9
pari, 2	pari, 6	pari, 10
JNTU, 2	SJ, 6	Earth, 10
tiru, 2	T.G, 6	tiru, 10
pari, 3	pari, 7	pari, 11
SJ, 3	SJS, 7	pari, 11
T, 3	T, 7	pari, 11
pari, 4	pari, 8	tiru, 12
CA, 4	USA, 8	pari, 12
Gandhi, 4	tirumali, 8	tiru, 12

Figure 14: Key value pairs generated by Map

pari, 1
pari, 2
pari, 3
pari, 4
pari, 5
pari, 6
pari, 7
pari, 8
pari, 9
pari, 10
pari, 11
pari, 11
pari, 11
pari, 12

SJSU, 1

tirumali, 1
tirumali, 8
tirumali, 9

JNTU, 2

tiru, 2
tiru, 10
tiru, 12
tiru, 12

SJ, 3
SJ, 6

T, 3
T, 7

Gandhi, 4

CA, 4
Ca, 5

T.G, 5
T.G, 6

SJS, 7

USA, 8

India, 9

earth, 10

Figure 15: Grouping similar entities

pari, <1,2,3,4,5,6,7,8,9,10,11,12>
SJSU <1>
tirumali <1,8,9>
JNTU <2>
tiru <2,10,12>
SJ <3, 6>
T <3,7>
CA <4,5>
Gandhi <4>
T.G <5,6>
SJS <7>
USA <8>
India <9>
earth <10>

Figure 16: Sorting the keys associated with each entity

The reducer creates following entity pairs

1-2, 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, 1-9, 1-10, 1-11, 1-12, 2-3, 2-4, 2-5, 2-6, 2-7, 2-8, 2-9, 2-10, 2-11, 2-12, 3-4, 3-5, 3-6, 3-7, 3-8, 3-9, 3-10, 3-11, 3-12, 4-1, 4-5, 4-6, 4-7, 4-8, 4-9, 4-10, 4-11, 4-12, 5-6, 5-7, 5-8, 5-9, 5-10, 5-11, 5-12, 6-7, 6-8, 6-9, 6-10, 6-11, 6-12, 7-8, 7-9, 7-10, 7-11, 7-12, 8-9, 8-10, 8-11, 8-12, 9-10, 9-11, 9-12, 10-11, 10-12, 11-12, <u>1-8, 1-9, 8-9, 8-10, 2-12, 10-12, 3-6, 3-7, 4-5, 5-6</u>

Figure 17: Entity pairs

Total number of pairs=76

Redundant pairs= 10

Efficiency improvement = (10/76)*100 = 13.15%

4. Implementation Details

This section describes about the algorithm implemented as part of baseline, the dataset chosen for experimentation and other tools and technologies used throughout the implementation of this project are explained.

The problem definition section has clearly described about the solution for how to avoid the redundant comparison. In this section I will describe about how I used Sparks libraries to solve this problem.

4.1 Baseline Implementation 1

In the paper referred the author talks about using the index of the string valued key instead of using the string itself. So as a part of the initial implementation, I implemented the algorithm using this technique. I created index called ID which is a unique value for every record present in the dataset and created a indexed file and the data is saved in the following format. For this implementation I used Medical Health contacts dataset and it had some 38 columns and most of them has null values or most of the columns had the same values for almost all the records. Using such data will not be of much help for this implementation. So I chose only few columns which are mostly not null. I chose Agency name, phone number, toll free number, email and web address columns.

[ID,[AGENCY, LOCALPHONE, TOLLFREEPHONE, EMAIL, WEB]]

The indexed file is generated as shown in the following figure and each spark partition can handle an RDD of 2GB. With this extraordinary feature of spark, we can give large amounts of data to process and spark does it very easily.

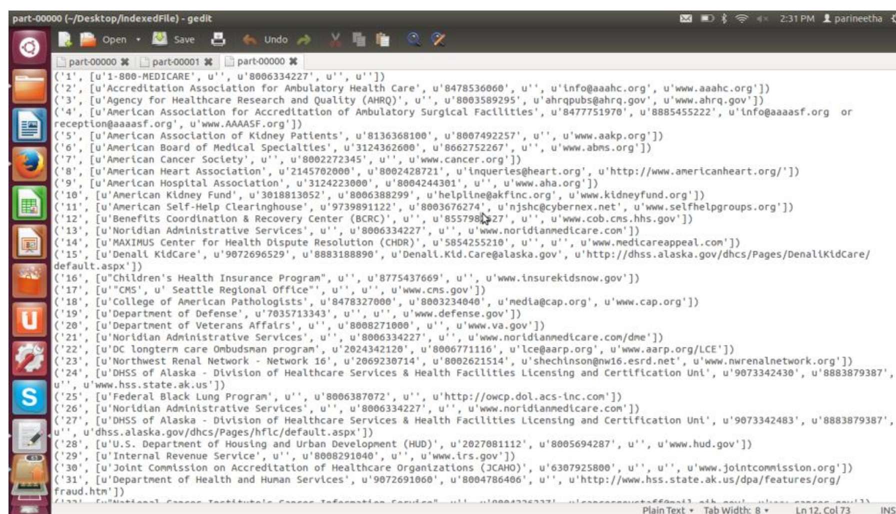


Figure 18: Snapshot of indexed file

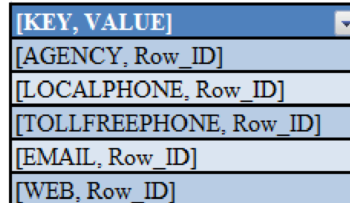
When there are multiple values in a row, Spark considers the first value as the key and all other values as values. As the first value(key) in the indexed file is ID, map function is used to swap the value with the key and make value as key. This was just an idea given by the author, implemented it to test for the match entities within a file and it worked. Following line of code is used to get the key value pairs as we needed

```
agency = indexedFile.mapValues(lambda x: x[0]).filter(lambda (u,v):
v!=’’).map(lambda (x,y): (y, x))

localPhone = indexedFile.mapValues(lambda x: x[1]).filter(lambda
(u,v): v!='').map(lambda (x,y): (y, x))

tfPhone = indexedFile.mapValues(lambda x: x[2]).filter(lambda (u,v):
v!='').map(lambda (x,y): (y, x))

email = indexedFile.mapValues(lambda x: x[3]).filter(lambda (u,v):
v!='').map(lambda (x,y): (y, x))

web = indexedFile.mapValues(lambda x: x[4]).filter(lambda (u,v):
v!='').map(lambda (x,y): (y, x))
```

Figure 19: Code snippet to generate key value pairs from map

Same filter operation is performed on all other keys such as LOCALPHONE, EMAIL, TOLLFREEPHONE, WEB.

And I generated the file as shown in the following table



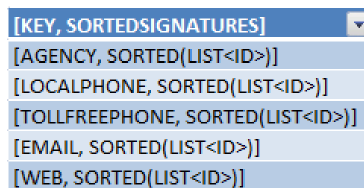
[KEY, VALUE]
[AGENCY, Row_ID]
[LOCALPHONE, Row_ID]
[TOLLFREEPHONE, Row_ID]
[EMAIL, Row_ID]
[WEB, Row_ID]

Figure 20: Snapshot of indexed file contents

Later entities having the same keys are grouped together and a list is generated to group entities having the same key

Using the groupbykey operation provided by spark on [AGENCY, Row_ID] and other attributes, I created a new dataset consisting of [AGENCY, Iterable<Row_ID>] and other attributes with their respective iterable value. After grouping all the attributes I sorted their values in ascending order using the following line of code.

```
matched_agencies = agency.groupByKey().mapValues(lambda x:
sorted(list(x)))
```



[KEY, SORTEDSIGNATURES]
[AGENCY, SORTED(LIST<ID>)]
[LOCALPHONE, SORTED(LIST<ID>)]
[TOLLFREEPHONE, SORTED(LIST<ID>)]
[EMAIL, SORTED(LIST<ID>)]
[WEB, SORTED(LIST<ID>)]

Figure 21: Snapshot of Entity, SortedSignatures

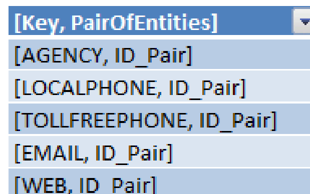
Generating entity pairs: To generate pairs, first element is taken and paired with rest of the elements until there's no element left in list. Following function is used to generate pair of IDs out of list of IDs

```
def pair(list):
        out = []
        while (len(list) > 0):
        popped = list.pop()
        for x in list:
                out.append(str(popped)+"-"+str(x))
        return out
```

Figure 22: Code snippet for generating entity pairs

Using above defined pair function to work on each sorted list of IDs. flatMapValues() generates the pairs of IDs out of list of IDs, and rather than returning list of pairs, puts each pair in new line.

```
flat_matched_agencies = matched_agencies.flatMapValues(lambda x:
pair(x))
```



[Key, PairOfEntities]
[AGENCY, ID_Pair]
[LOCALPHONE, ID_Pair]
[TOLLFREEPHONE, ID_Pair]
[EMAIL, ID_Pair]
[WEB, ID_Pair]

Figure 23: Snapshot of Key value pairs in the form of key and pairs of entities

This was a trial implementation of first baseline. The output of it is shown in the following figure.

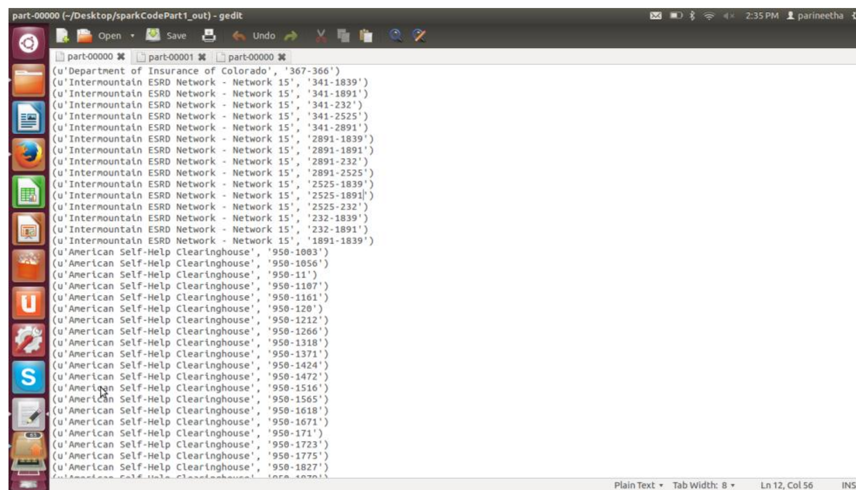


Figure 24: Generation of Entity Pairs

4.2 Baseline Implementation 2

When implementing the algorithm on big data, it is very important to check the quality of the data. In the dataset I chose there were many data quality issues like

New line characters: When I ran the code I was not aware of the new line characters present in few records. Debugging and figuring out what the problem was very time consuming. There were very few attributes which had new line character, so to retain the consistency these records were deleted.

Null values: As the dataset is very large it is expected to have null values. There were many fields in the dataset which had 90% of the records as nulls. Considering such fields would just create lag in the execution time with no positive effect on the result. So these fields were removed.

Baseline implementation 2 is more Spark oriented. The main goal of this paper is to implement efficient pair-wise similarity computation using Spark. To implement the algorithm suggested in the paper I used the concept of Data Frames and Windows from the Spark libraries.

Input: Medical Health dataset with 996,000 records.

Output: Output generated two files named eliminated and matched. Matched file has all the pair of entities that matched only once. Eliminated file has all the pairs of records that were supposed to be matched redundantly.

Made use of following pyspark libraries
pyspark.sql.Window: A distributed collection of data grouped into named columns.
pyspark.sql.Row: A row of data in a DataFrame.
pyspark.sql.HiveContext: Main entry point for accessing data stored in Apache Hive

4.2.1 Hive Context: It is a superset of SQL Context as it provides all the functionalities of SQL and Hive as well.

4.2.2 Data Frames: Data frames in spark provide the flexibility of collecting the data from various data sources such as resilient distributed dataset or external files or Hive tables, etc. In simple terms it is like creating a table with named attributes. To create a data frame we first created the schema, column names and their datatypes.

- Generate a new RDD out of list/ tuple of already existing RDDs

aoPair = inRDD.flatMap(lambda line: attr_key(line.split("\t")))

- Define the schema for the data frame. Spark has become so advanced that if the datatype of the column is not mentioned then it will try to infer from the data what type it can be by going through some amount of the data which is called sampling ration[11].

schema = StructType([StructField("attr", StringType(), True), StructField("obj", StringType(), True)])

class `pyspark.sql.types.StructField`(*name, dataType, nullable=True, metadata=None*)

A field in `StructType`.

Parameters:
- **name** – string, name of the field.
- **dataType** – `DataType` of the field.
- **nullable** – boolean, whether the field can be null (None) or not.
- **metadata** – a dict from string to simple type that can be toInternald to JSON automatically

- Finally using the createDataFrame function apply the above schema to the RDD

aoDF = sqlCtx.createDataFrame(aoPair, schema)

attr	obj
1	a
2	a
3	a
1	b
2	b
3	b
1	c
3	c

Figure 25: List of attributes and objects

4.2.3 Window Concept in Spark

Once the data frame is created with the mentioned schema, we need to perform operations on this data frame. I used the concept called Windows in Spark. Apache Spark allow us to perform certain functionalities on group of rows. These group of rows is known as window. By defining a window we can perform some operations on data frames[11]. Spark SQL provides 3 kinds of aggregate functions on windows: Ranking, Analytic and Aggregate functions. window object is in the package called pyspark.sql, So we need to import it using the following line of code

from pyspark.sql import HiveContext, Row, Window

memorize = aoDF.select("attr", "obj", lag("attr",1, None).over(window).alias("prev"))

The lag method gets the previous records for the current record and the parameter 1 represents get it from the one previous row and the parameter None represents what to do when there is no previous value. over(window) defines a windowing column and to jumps over one window at a time. If the current window is for the obj "a" then it process the window 'a' and then only jumps to the other window and process that. The alias function returns a new data frame and in this case we are creating a new data frame called "prev".

attr	obj	prev
1	a	null
2	a	1
3	a	2
1	b	null
2	b	1
3	b	2
1	c	null
3	c	1

Figure 26: List of attributes, objects and previous values

Finally the data frame created looks like

DataFrame : [attr, obj, prev] → RDD : [(attr, (obj, prev))]

attr	(obj, prev)
1	(u'a', None)
2	(u'a', 1)
3	(u'a', 2)
1	(u'b', None)
2	(u'b', 1)
3	(u'b', 2)
1	(u'c', None)
3	(u'c', 1)

Figure 27: List of attributes and object, previous pairs

mappedRDD = memorize.map(lambda row: (row.attr, (row.obj, row.prev)))

groupedByAttr = mappedRDD.groupByKey().mapValues(list).cache()

[(1, [(u'a', None), (u'b', None), (u'c', None)]), (2, [(u'a', 1), (u'b', 1)]), (3, [(u'a', 2), (u'b', 2), (u'c', 1)])]

Later grouped it by attribute and collected the tuple (obj, prev) into the list

Implemented following algorithms by making use of the spark libraries

```
map(k_in=unused, v_in=o)
    S ← (σ).distinct(); //calculates the distinct keys
    S.sort();      //makes use of sorted signature list
    SS ← [ ]       // smaller signature list
    foreach s_i ∈ S do
        output(k_tmp = si, v_tmp= (o, SS));
        SS.append(s_i);
```

Figure 28: Algorithm for Map Phase

```
reduce(k_tmp=s, list(v_tmp)=list(object, SS))
    buf ← { };
    foreach (σ_1, SS_1) ∈ list(object, SS) do
        foreach (o_2, SS_2) ∈ buf do
            if doOverlap (SS_1, SS_2) then
                compare(o_1, o_2);
        buf←buf ∪ {(o_1, SS_1)};
```

Figure 29:Algorithm for Reduce Phase

```
doOverlap(SS_1, SS_2)
    i←0;
    j←0;
    l_1←SS_1.length(); //object 1's sorted signature list
    l_2←SS_2.length(); //object 2's sorted signature list

    while (i < l_1) ∧ (j < l_2) do // find the signatures
which are common
        s1←SS1.get(i);
        s2←SS2.get(j);
        cmp←s1.compareTo(s2); //compareTo returns 0 if
s1 and s2 are same.
        if cmp = 0 then
            return true;
        else if cmp < 0 then
            i++;
        else
            j++;
    return false;
```

Figure 30:Algorithm for Overlap

5. Performance Evaluation

I have performed the implementation on Spark cluster on virtual machines. This set up had 3 nodes with one as master node and two others as slave nodes. Each node is of 2GB RAM and installed with Ubuntu 14.04. Experimentation is done using Apache Spark version 1.4 with 2.4 Hadoop distribution. The data frames concept in spark was introduced in the version 1.3, so I had to use version 1.3 or beyond for this implementation.

Tested the implementation in the Spark standalone cluster mode on Medical Health dataset called Plaid which had 996000 records. Without the implementation of this algorithm the simple Cartesian product have done matching in very naive, compare one record with each other record present in the dataset. With this implementation we have narrowed down the search space and reduced the comparisons 163,527. We get 119,369 records as matching and eliminated 44158 matches which reduced the number of comparisons by 44158/163527 = 27% of the records are eliminated as redundant.

Execution time with and without redundant pairs

Number of records	Execution time with redundant pairs(minutes)	Execution time without redundant pairs(minutes)
100k	34.1	31.7
300k	68.8	65
500k	166.6	105.4
700k	158	149.8
996k	225.8	213.6

Table 1: Execution time with and without redundant pairs

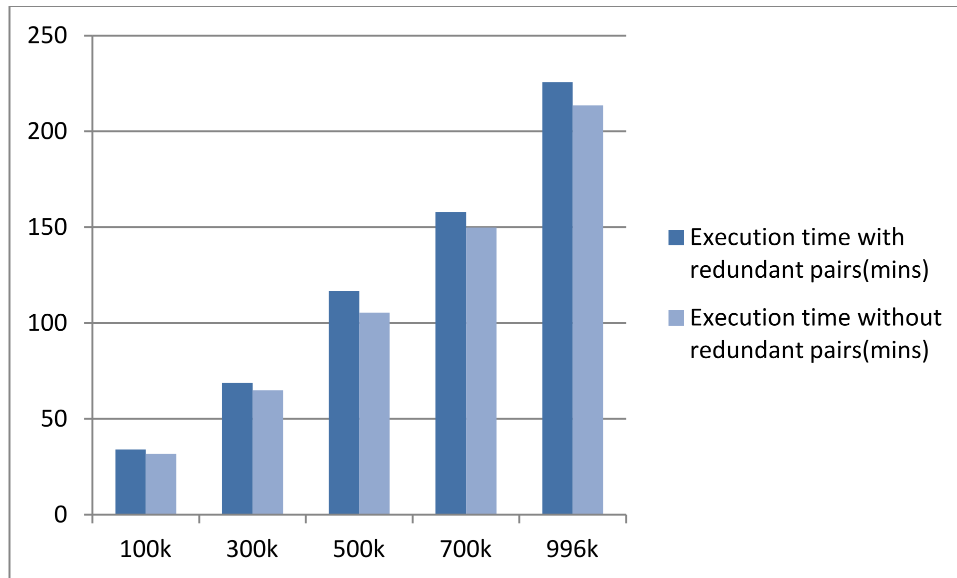


Figure 31: Execution times comparison

5.1 Apache Spark Cluster Setup

Implementation is done in Virtual Machine mode where using VM Workstation and created two virtual machines with Ubuntu version 1.4 installed. Configured these VMs with 2 GB RAM, 20GB hard disk and quad core processor.

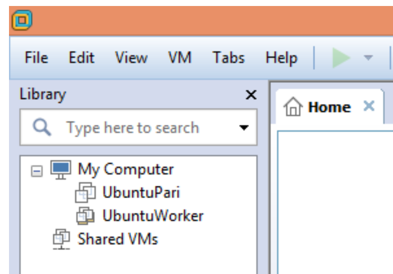


Figure 32: Screenshot of two VMs in VMWare WorkStation

Install Java on the VM and update the JAVA_HOME in environmental variables

JAVA_HOME=/usr/lib/jvm/java-6-oracle/

```
pari@ubuntu:~$ sudo nano /etc/environment
pari@ubuntu:~$ sudo nano /etc/environment
pari@ubuntu:~$ java -version
java version "1.6.0_45"
Java(TM) SE Runtime Environment (build 1.6.0_45-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.45-b01, mixed mode)
pari@ubuntu:~$ source /etc/environment
pari@ubuntu:~$ echo $JAVA_HOME
/usr/lib/jvm/java-6-oracle/
pari@ubuntu:~$
```

Figure 33: Screenshot showing Java version on VM

Download debian version of Scala if you are using Ubuntu

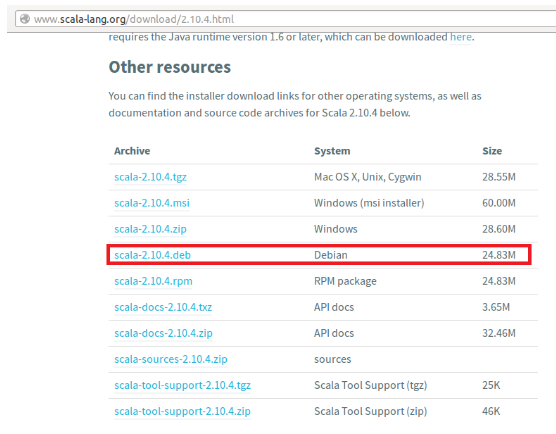


Figure 34: Screenshot for Scala download

and install it from the software center. I installed 2.10.4 version

```
pari@ubuntu:~$ scala -version
Scala code runner version 2.10.4 -- Copyright 2002-2013, LAMP/EPFL
pari@ubuntu:~$
```

Figure 35: Screenshot showing Scala version on VM

Clone the master node to get the virtual machine with all the softwares installed till now. With few modification we set the second virtual machine as Worker node.

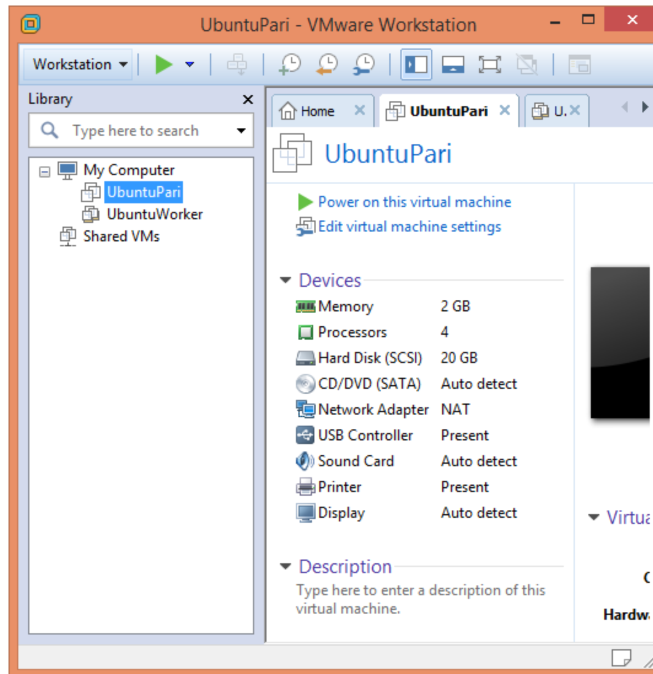


Figure 36: Screenshot of Master Node

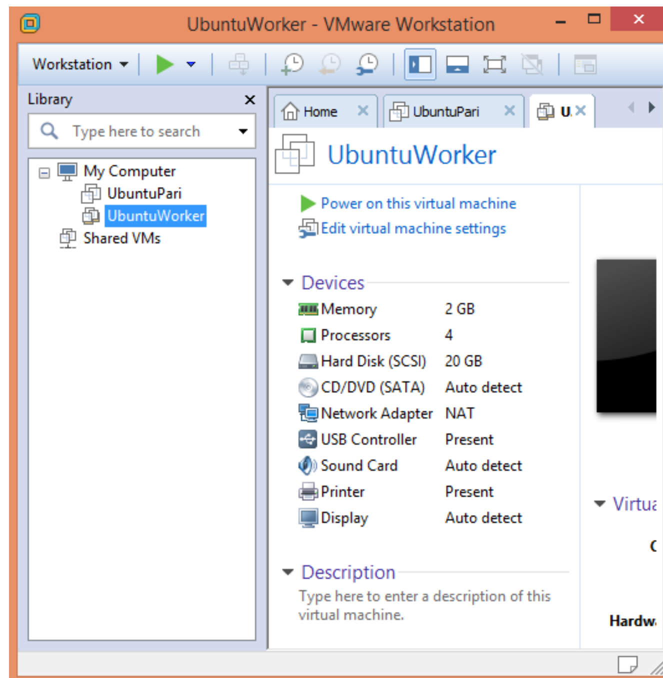


Figure 37: Screenshot of Worker Node

Once both machines are ready, install ssh on worker node to give access to the master node to access worker.

```
pari@ubuntu:~$ sudo apt-get install openssh-server
[sudo] password for pari:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
```

Figure 38: Screenshot for installing ssh

Later RSA key needs to be generated on the master node to obtain remote access. Connect the master node with the worker node. Copy the public key generated earlier to each worker node. This gives master to access worker node with SSH

```
pari@ubuntu:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub pari@192.168.77.129
The authenticity of host '192.168.77.129 (192.168.77.129)' can't be established.
ECDSA key fingerprint is a0:09:ef:d3:f4:4e:10:38:84:f6:e1:34:1a:9d:72:84.
Are you sure you want to continue connecting (yes/no)? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompt
ed now it is to install the new keys
pari@192.168.77.129's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'pari@192.168.77.129'"
and check to make sure that only the key(s) you wanted were added.

pari@ubuntu:~$ ssh 'pari@192.168.77.129'
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

88 packages can be updated.
```

Figure 39: Screenshot showing Master connection

Create a new VM and repeat the same process and take screenshots

Download and install required version of spark on both the nodes.

Download Apache Spark™

Our latest version is Spark 1.6.1, released on March 9, 2016 (release notes) (git tag)

1. Choose a Spark release: 1.4.0 (Jun 11 2015)
2. Choose a package type: Pre-built for Hadoop 2.6 and later
3. Choose a download type: Direct Download
4. Download Spark: spark-1.4.0-bin-hadoop2.6.tgz
5. Verify this release using the 1.4.0 signatures and checksums.

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

Figure 40: Spark version

Until now these two nodes are the same as we have not given master or worker specifications to any node. The main settings should be made in the conf file of the spark package. conf file consists of following files

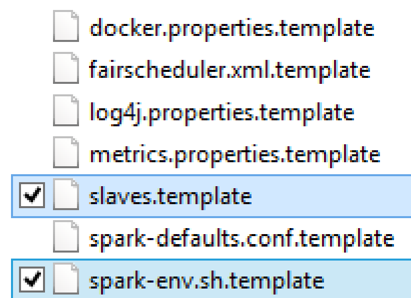


Figure 41: Files in conf folder

Make a copy of slaves.template file and spark-env.sh.template file.

Rename "slaves.template(copy)" to "slaves" and "spark-env.sh.template(copy)" to "spark-env.sh".

Add the IP address of the worker nodes in the slaves file.

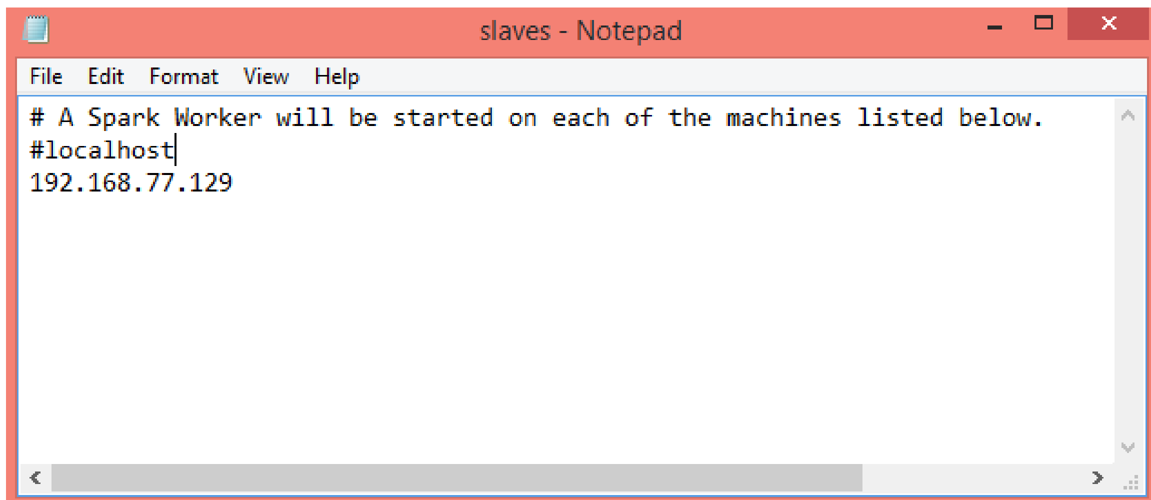


Figure 42: Updated Slaves file

Add the following lines in spark-env.sh file.

export SPARK_MASTER_IP=192.168.77.130

export SPARK_WORKER_CORES=1

export SPARK_WORKER_MEMORY=800m

export SPARK_WORKER_INSTANCES=2

```
31 # Options for the daemons used in the standalone deploy mode
32 # - SPARK_MASTER_IP, to bind the master to a different IP address or hostname
33 export SPARK_MASTER_IP=192.168.77.130
34 # - SPARK_MASTER_PORT / SPARK_MASTER_WEBUI_PORT, to use non-default ports for the master
35 # - SPARK_MASTER_OPTS, to set config properties only for the master (e.g. "-Dx=y")
36 # - SPARK_WORKER_CORES, to set the number of cores to use on this machine
37 export SPARK_WORKER_CORES=1
38 # - SPARK_WORKER_MEMORY, to set how much total memory workers have to give executors (e.g. 1000m, 2g)
39 export SPARK_WORKER_MEMORY=800m
40 # - SPARK_WORKER_PORT / SPARK_WORKER_WEBUI_PORT, to use non-default ports for the worker
41 # - SPARK_WORKER_INSTANCES, to set the number of worker processes per node
42 export SPARK_WORKER_INSTANCES=2
43 # - SPARK_WORKER_DIR, to set the working directory of worker processes
44 # - SPARK_WORKER_OPTS, to set config properties only for the worker (e.g. "-Dx=y")
45 # - SPARK_HISTORY_OPTS, to set config properties only for the history server (e.g. "-Dx=y")
46 # - SPARK_SHUFFLE_OPTS, to set config properties only for the external shuffle service (e.g. "-Dx=y")
47 # - SPARK_DAEMON_JAVA_OPTS, to set config properties for all daemons (e.g. "-Dx=y")
48 # - SPARK_PUBLIC_DNS, to set the public dns name of the master or workers
```

Figure 43: Updated spark-env.sh file

To test the installation, run

./bin/spark-shell

scala:> sc.parallelize(1 to 1000).count();

(This should give the output as long=1000)

scala:>exit

./bin/run-example SparkPi

(This should give the output of calculated pi value)

5.2 Launching and testing the cluster

The sbin folder has the files to start or stop a master or slave

To start the cluster we need to execute following command in the terminal

./sbin/start-all.sh

and to stop

./sbin/stop-all.sh

Name

slaves.sh
spark-config.sh
spark-daemon.sh
spark-daemons.sh
start-all.sh
start-history-server.sh
start-master.sh
start-mesos-dispatcher.sh
start-shuffle-service.sh
start-slave.sh
start-slaves.sh
start-thriftserver.sh
stop-all.sh
stop-history-server.sh
stop-master.sh
stop-mesos-dispatcher.sh
stop-shuffle-service.sh
stop-slave.sh
stop-slaves.sh
stop-thriftserver.sh

Figure 44: Files in sbin folder

6. Conclusion

This project implements pair-wise similarity computation without redundancy using Apache Spark. The implementation is done successfully on large dataset consisting of 996000 records. As Apache spark is a powerful big data processing engine which has the concept of windows and data frames using which we tried improving this area of entity matching has scope for more improvement. Improving the concept of redundant comparison is a very rare study. This project can be further improvised by implementing multiple blocking strategies. It can be implemented using Amazons cluster and try it for more larger dataset. This algorithm with further research and slight modifications can also be tested upon the datasets consisting of images, text and videos.

7. References

[1] Hanna Köpcke, Erhard Rahm, Frameworks for Entity Matching: A comparison. Database Group, University of Leipzig, Postfach 100920, 04009 Leipzig, Germany, Journal Data & Knowledge Engineering, volume 69, issue 2, pages 197-210, 2009 http://dl.acm.org/citation.cfm?id=1672420

[2] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl: Eliminating the Redundancy in Blocking-based Entity Resolution Methods, In Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries, pages 85-94, 2011 http://dl.acm.org/citation.cfm?id=1998093&dl=ACM&coll=DL&CFID=619571752 &CFTOKEN=36873886

[3] Lars Kolb, Andreas Thor, Erhard Rahm: Don't Match Twice: Redundancy-free Similarity Computation with MapReduce, In Proceedings of the Second Workshop on Data Analytics in the Cloud, pages 1-5, 2013 http://dl.acm.org/citation.cfm?id=2486768

[4] Jimmy Lin: Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce, Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, pages 155-162, 2009 http://dl.acm.org/citation.cfm?id=1571970

[5] Groß, A., Hartung, M., Kirsten, T., Kolb, L., Köpcke, H., & Rahm, E. Data Partitioning for Parallel Entity Matching. CoRR, abs/1006.5309, 2010.

[6] Lars Kolb, Erhard Rahm: Parallel Entity Resolution with Dedoop, volume 13, issue 1, pages 23-32, Journal Datenbank- Spektrum, 2013, http://link.springer.com/article/10.1007/s13222-012-0110-x

[7] [Online, accessed on January 2015] Apache Spark programming guide: http://spark.apache.org/docs/latest/programming-guide.html

[8] [Online, accessed on August 2015] Installing Apache spark on standalone cluster mode: http://www.trongkhoanguyen.com/2014/11/how-to-install-apache-spark-121-in.html

[9] [Online, accessed on August 2015] PySpark Guide: https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

[10] [Online, accessed on August 2015] pyspark.sql.DataFrame and window Guide: http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFram e

Image References:

1. [Online, accessed on February 2016] http://studybigdata.co/data-flow/
2. [Online, accessed on February 2016] http://docplayer.net/6327218-Hadoop-mapreduce-and-spark-giorgio-pedrazzi-cineca-scai-school-of-data-analytics-and-visualisation-milan-10-06-2015.html