

Fall 2015

# LOAD BALANCING FOR BIG DATA ENTITY MATCHING USING BLOCK SPLIT

Akhilesh Kondra  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_projects](https://scholarworks.sjsu.edu/etd_projects)

Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Kondra, Akhilesh, "LOAD BALANCING FOR BIG DATA ENTITY MATCHING USING BLOCK SPLIT" (2015). *Master's Projects*. 457.

DOI: <https://doi.org/10.31979/etd.jez2-jkmx>

[https://scholarworks.sjsu.edu/etd\\_projects/457](https://scholarworks.sjsu.edu/etd_projects/457)

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](mailto:scholarworks@sjsu.edu).

LOAD BALANCING FOR BIG DATA ENTITY MATCHING USING  
BLOCK SPLIT

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  
Akhilesh Kondra  
Dec 2015



The Designated Project Committee Approves the Project Titled

LOAD BALANCING FOR BIG DATA ENTITY MATCHING USING  
BLOCK SPLIT

by  
Akhilesh Kondra

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

Dec 2015

Dr. Tran Duc Thanh      Department of Computer Science.

Dr. H. Chris Tseng      Department of Computer Science.

Mr. Venkat Vattikuti      Manager, Servicenow.

**ABSTRACT**  
**LOAD BALANCING FOR BIG DATA ENTITY MATCHING USING**  
**BLOCK SPLIT**

by Akhilesh Kondra

Entity Matching (EM) is a complex problem and has great impact on data quality. In EM we usually match all the combination of entity pairs using different similarity measures and judge if there is any match between entities. Mapreduce based parallel programming model can be used to match these entities. Even distribution of data into the map and reduce tasks will play vital role in the productivity of Mapreduce based programming model. If the dataset is large and has skewed data, then the distribution should be done effectively to achieve load balancing.

In this paper, I have implemented an approach of blocking technique called “Block Split”. Block split will reduce the search space of match tasks by splitting larger blocks into multiple small blocks and process it using mapreduce model. This approach utilizes two mapreduce jobs, one to identify the data distribution in each block and use this distribution to perform the match tasks in the second job. The effectiveness of block split approach is described in terms of ‘recall’ and ‘precision’. To improve recall I iteratively applied blocking of different keys by assigning every input record to different blocks (one per blocking key) and then found matches per blocks. Using this we will most likely find more matches but, we may come across many redundant matches. I have optimized the above approach by using “Signature Based Pair Comparison”. We evaluated all our approaches on spark clusters.

## ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Tran Duc Thanh, for his continuous guidance and support throughout the project and providing me an opportunity to work on this project. I would also like to thank my committee members, Dr. H. Chris Tseng and Mr. Venkat Vattikuti, for their valuable time and feedback. Lastly, I would also like to convey my thanks to my family, and friends for their help and support.

# Table of Contents

## CHAPTER

<b>1. Introduction</b>	<b>6</b>
<b>2. Related Works</b>	<b>7</b>
2.1 Spark Execution Framework	9
<b>3. Problem Definition</b>	<b>10</b>
<b>4. Proposed Solution</b>	<b>12</b>
4.1 Block Distribution Matrix (BDM):	13
4.2 Block Split - Match Task:	15
4.3 Match Algorithms:	19
<b>5. Implementation</b>	<b>21</b>
5.1 Spark Cluster Setup:	21
5.2 Installations and Configuration	24
5.2 Sample Code	25
5.3 Execution Report	29
<b>6. Performance Evaluation</b>	<b>31</b>
6.1 Degree of Data Skew	31
6.2 Block Split blocking evaluation in terms of Precision and Recall:	32
6.2.1 Optimization	33
6.3 Signature Based Entity Comparison:	34
6.3 Increase in number of Reduce Tasks and nodes	38
<b>7. Conclusion</b>	<b>39</b>
<b>8. References</b>	<b>40</b>

## LIST OF FIGURES

Figure 1: Duplicate product entries	6
Figure 2 : MapReduce execution framework	8
Figure 3: Spark execution framework	9
Figure 4: To demonstrate how blocking works	10
Figure 5 : Workflow for mapreduce based blocking techniques using BDM	13
Figure 6 : Algorithm for Block Distribution Matrix (BDM)	14
Figure 7: Example set of 14 entities with 4 blocking keys	14
Figure 8 : MapReduce workflow for calculating BDM	15
Figure 9 : BDM generated from the example of 14 entities	17
Figure 10 : MapReduce workflow for Block Split	17
Figure 11 : Algorithm for Block Split Map task	18
Figure 12 : Algorithm for Block Split Reduce task	19
Figure 13 : Edit distance algorithm implemented in Java	20
Figure 14 : Set of Instructions to install JAVA SDK on to VM	21
Figure 16 : Set of Instruction to configure hadoop file system	22
Figure 17: Set of Instructions to Download and Install Spark on VM cluster	23
Figure 18: Spark Web UI context with nodes information	24
Figure 19 : Spark Code to create spark context	25
Figure 20 : Spark code to implement BDM	26
Figure 21 : Spark code to implement map and reduce function	27
Figure 22 : Sample Output of Entity Matching after Block Split	28
Figure 23 : Evaluation report for Block Split execution with different data skew factors	32
Figure 24 : Algorithm to implement Signature Based Entity Matching	35
Figure 25 : Set of entities to multiple key blocking	36
Figure 26 : Sample Entity Matching Output after Signature Based Entity Matching	36
Figure 27 : Workflow for Signature Based Entity Matching with Block Split	37



## CHAPTER 1

### Introduction

Mapreduce (MR) model is one among many programming models that facilitates the parallel execution of complex task like entity matching. The timelessness and the productivity of a mapreduce implementation, completely depends on an effective way of the balancing the execution load between the available nodes. Especially, this is very challenging for skewed data, as it may result in bottlenecks and also causes load imbalances problems on the node under execution.

MR model can also be used to implement effective Entity matching (EM) .EM is also known as entity resolution, in which we determine all entities (duplicates) referring to the same real world object from given a set of data sources. Some examples of the entity matching tasks are to find duplicate employees or customers or products in the company database or to match the price and discounts for a product published by different vendors.







	<b>Canon VIXIA HF S10 Camcorder - 1080p - 8.59 MP - 10 x optical zoom</b> 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 - <a href="#">Add to Shopping List</a>	<b>\$975</b> new from 52 sellers  <a href="#">Compare prices</a>
	<b>Canon (VIXIA) HF S10 iVIS Dual Flash Memory Camcorder</b> Canon HF S10 iVIS Dual Flash Memory CamcorderSPECIAL SALE PRICE: \$899 Display both English/Japanese + we supplu all English manuals in English as PDF. ... <a href="#">Add to Shopping List</a>	<b>\$899.00</b> new Made in Japan Online
	<b>Canon VIXIA HF S10</b> 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, ... <a href="#">Add to Shopping List</a>	<b>\$999.00</b> new Performance Audio <a href="#">2 seller ratings</a>
	<b>Canon VIXIA HF S100 Flash Memory Camcorder</b> ***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 ... <a href="#">Add to Shopping List</a>	<b>\$899.95</b> new Arlingtoncamera.com <a href="#">5 seller ratings</a>
	<b>Canon Vixia Hf S10 Care &amp; Cleaning</b> Care & Cleaning Digital Camera/Camcorder Deluxe Cleaning Kit with LCD Screen Guard Canon VIXIA HF S10 Camcorders Care & Cleaning. <a href="#">Add to Shopping List</a>	<b>\$2.99</b> new shop.com ★★★★☆ 38 seller ratings

Figure 1: Duplicate product entries

## **CHAPTER 2**

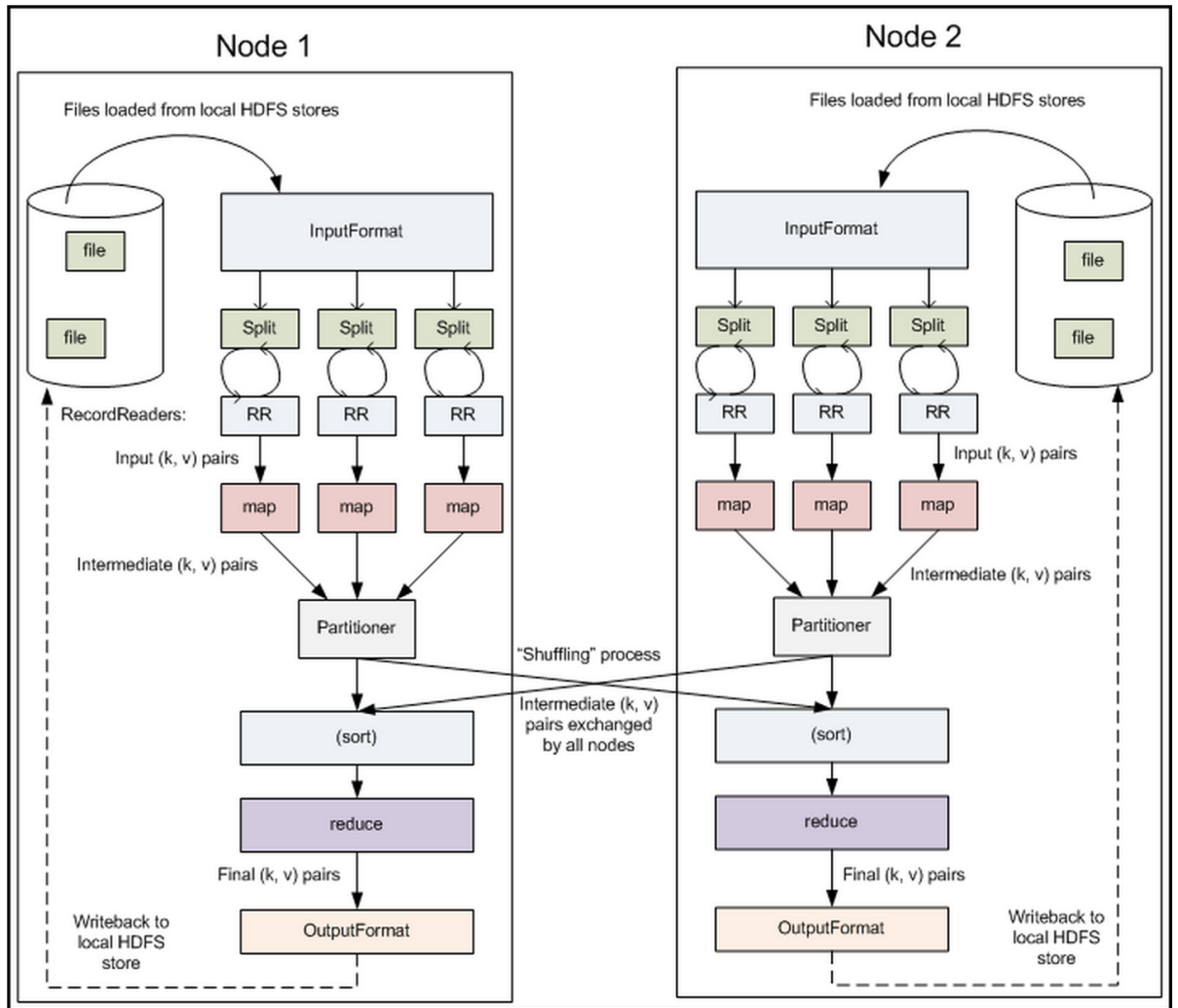
### **Related Works**

Parallel programming is playing a vital role in solving the complex task by splitting the tasks into simple and easy for processing. The best example for parallel programming is implementation of mapreduce model. But in the mapreduce model the biggest unhandled problem is that it cannot handle load for skewed data. This is because the processor running the map and reduce tasks for skewed data takes more than expected execution time and ends up blocking other tasks. My work will concentrate on handling the skewed data and load imbalances.

Understanding the Mapreduce framework: It makes crucial task to understand the complete workflow of a mapreduce framework in order to solve the load imbalance issue.

**The figure below depicts the complete flow of data and execution for a MapReduce job in two nodes.**

1. Data will be first loaded from any file system
2. The job defines the input format of the data
3. Data is split between different map() methods running on all the nodes
4. Key value pairs are generated by parsing the data using the Record readers that will now serve as input into the map() function.
5. The map() method produces key value pairs that are sent to the partition.
6. When there are multiple reducers, the mapper creates one partition for each reduce task.
7. The key value pairs are sorted by key in each partition
8. The reduce() method takes the intermediate key value pairs and reduces them to a final list of key value pairs.
9. The job defines the output format of the data



**Figure 2 : MapReduce execution framework**

### **Implementation of MapReduce model using Spark**

Spark has the advantage of processing the data in memory when compared to hadoop mapreduce, which sends the data back to disk after every map and reduce function, this way spark will outperform when compared with hadoop. I had implemented the parallel processing of entity matching using Spark.

## 2.1 Spark Execution Framework

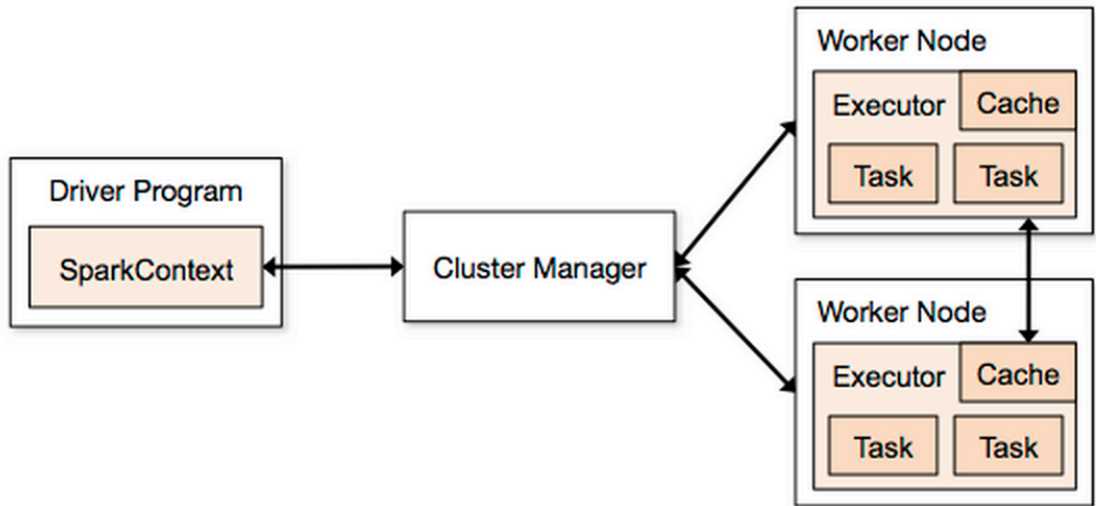


Figure 3: Spark execution framework

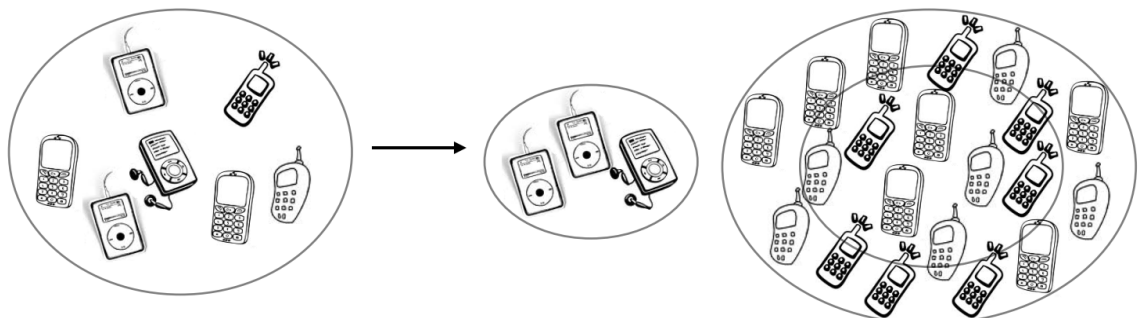
1. Each application gets its own executor processes, which persist throughout the application and runs in their own threads.
2. Spark is agnostic of the underlying cluster manager. Spark supports standalone mode, local mode by running in a YARN cluster.
3. Jobs are submitted to Spark using the spark submit script. One of the options to this script is the underlying cluster manager. The driver program runs the main () program and creates the spark context.

## CHAPTER 3

### Problem Definition

It is **hard to solve entity-matching problems over Big Data** because in EM we usually match all the combination of entity pairs using different similarity measures and judge if there is any match between entities. Naive approach leads to compare each entity with all other entities i.e. the Cartesian product of all the input entities. The complexity is  $O(n^2)$ . For larger datasets this cannot be achieved even in cloud infrastructure. The best way to improve productivity is by limiting the search space by implementing “Blocking Techniques”

**Blocking Techniques:** For larger sets of data input, it is very hard to perform entity comparison. Blocking will help to reduce the search space and group similar entities within blocks based on blocking algorithm, thus results in a smaller subset of entities where comparison needs to be happened. Many blocking algorithms exist but one of the most effective in the past years is using a key to partition the entities into blocks, here I refer the key as blocking key. Using a blocking key we can restrict and refrain entities to be matched to a smaller set of group. The key can be anything that can group entities, for example entities from a product database can be grouped by the manufacture.



**Figure 4: To demonstrate how blocking works**

### **There are two methods of blocking**

1. **Disjoint:** In this method we build mutually exclusive blocks, that is each input record or entity will be assigned to only one block, This method is good as it has less number of redundant comparison.
2. **Overlapping blocks:** In this method we build overlapping blocks. I.e. each input record can exists in different blocks. Using this method a better recall is achieved but many redundant comparisons

Finding out a perfect blocking key will play a vital role in entity comparisons, a wrong selection may result of dissimilar entities and selecting a blocking can be done either manually or in a semi-automated way using machine learning.

Despite use of blocking, Entity matching will remain costly in execution time and might end up processing for many hours or days because of the data skew blocks. Since the matching task will run for all the entities inside the block, load imbalance will occur because of the skewed data blocks. For example imagine a block containing (25%) of all the entities inside it, the node running the match task will continuously execute until matching all the entities are completed and makes other nodes with a small set of entities sit idle. The absence of a better approach to handle the data skews mechanisms will result in huge increase in execution time.

## **CHAPTER 4**

### **Proposed Solution**

In this paper I have implemented Block Split approach, an effective way of handling the above mentioned data skew problem with blocking techniques. In Block Split we will process all the smaller blocks with in one match task i.e. single map and reduce task for each small blocks and in the other hand we will split larger blocks into group of small blocks and distribute the blocks into different match tasks. Block Split divides the larger blocks into “p” number of sub blocks based on the number of input partitions. Now these p sub blocks will act like un-split block and perform the match task for each sub block and the pairs of these sub blocks are processed by another match task by performing a Cartesian produce of these sub blocks. This way we can ensure that all the comparison entities with the original blocks are computed.

Block split will first define the number of entity comparison per match task using Block Distribution Matrix and now assigns these match tasks in the decreasing order of size among the reduce task. This way we can assure that the largest match task will process first.

#### **Idea**

EM using block split processing in two MR jobs based on the same partitioning of the input data

- 1. Analysis job** – computation of the BDM that specifies the number of entity pairs per block separated by input partitions
- 2. Match job** – utilization of the BDM for load balancing strategies (e.g. Block Split) during the map phase & matching of entities in reduce phase

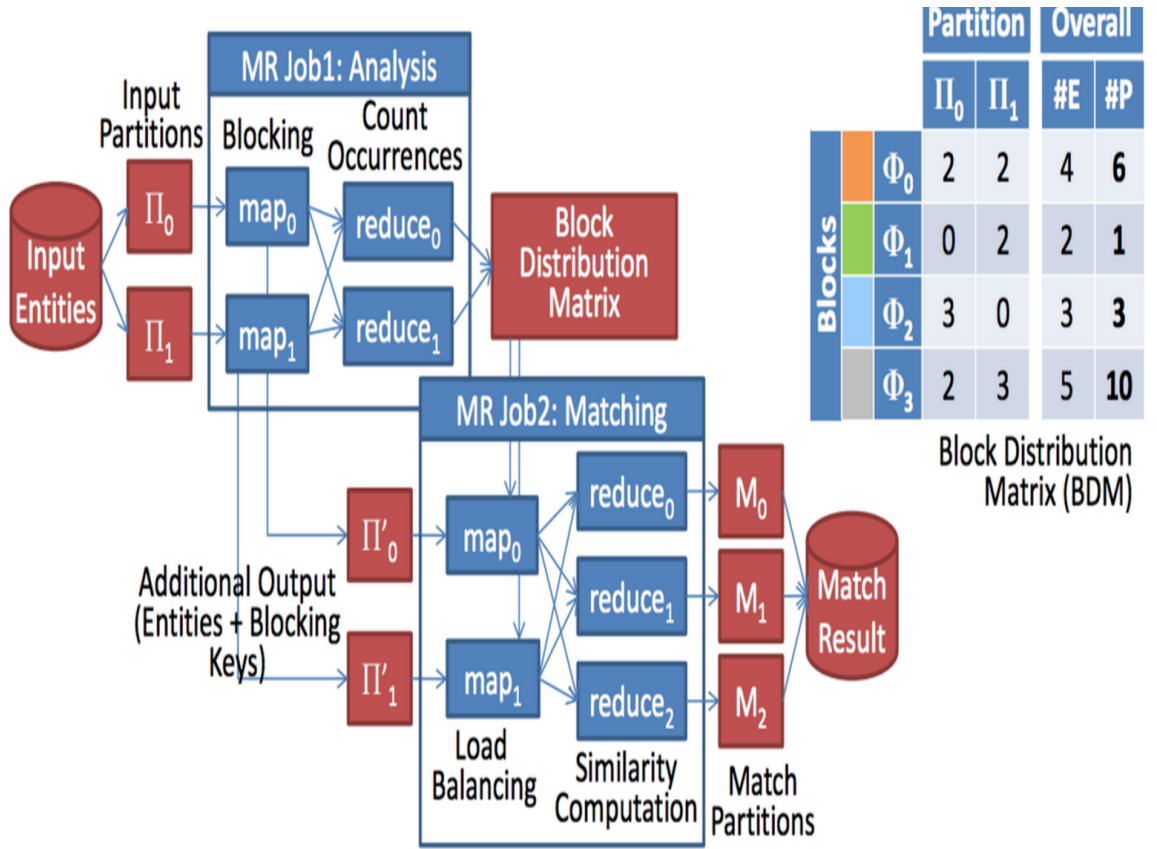


Figure 5 : Workflow for mapreduce based blocking techniques using BDM

#### 4.1 Block Distribution Matrix (BDM):

It is a matrix of  $b \times n$  in which  $b$  is the number of blocks and  $n$  is the number of input partitions. BDM is implemented using a simple map and reduce job. Map function generates key value pairs for each entity where key is (blockingkey.partitionindex) and the value is 1 for each entity. In the reduce task, pairs are sorted and grouped by the keys and counts the value. The output of the reduce task is a triplet [blocking key, partition index, and count of entities in the block]



## Algorithm for BDM Computation using Spark

```

SparkConf sparkConf = new SparkConf().setAppName("computeBDM");

JavaSparkContext ctx = new JavaSparkContext(sparkConf);

JavaRDD<String> lines = ctx.textFile(args[0], 1);

JavaRDD<String> entities = lines.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String s) {
        return Arrays.asList(SPACE.split("\n"));
    }
});

map configure(m, r, partitionIndex); // to Store partitionIndex

JavaPairRDD<String, String> map = entities.mapToPair (kin=unused, vin=entity)
{
    blockingKey = computeKey(entity);
    additionalOutput (k=blockingKey, v=entity) ; // to DFS
    output (ktmp=blockingKey.
, vtmp=1);
}

// Repartition map output by blockingKey, sort by
// blockingKey.partitionIndex, group by blockingKey.partitionIndex

JavaPairRDD<String,Integer> reduce = map.reduceByKey (ktmp=blockingKey.partitionIndex, list(vtmp)=list(1))
{
    sum ← 0;
    foreach number in list(vtmp) do
    {
        sum ← sum+number;
    }
    output (kout=unused, vout=blockingKey +","+partitionIndex +","+sum);
}

```

**Figure 6 : Algorithm for Block Distribution Matrix (BDM)**

For example, below figure depicts the computations of BDM for illustration purposes, we use a running example with 14 entities and 4 blocking keys as shown in Figure.

Partition	$\Pi_0$								$\Pi_1$					
Entity	A	B	C	D	E	F	G	H	I	K	L	M	N	O
BlockingKey	w	w	x	x	x	z	z	w	w	y	y	z	z	z

**Figure 7: Example set of 14 entities with 4 blocking keys**



$$P = \frac{1}{2} \cdot \sum_{k=0}^{b-1} |\Phi_k| \cdot (|\Phi_k| - 1)$$

Now for each blocks it also checks if the total number of comparison is above the average reduce workload.

$$\frac{1}{2} \cdot |\Phi_k| \cdot (|\Phi_k| - 1) > P/r.$$

If the total number of comparison per block is not above the average workload of reduce task then all the entities inside the block can be processed with one match task and map function key is (reduce\_index.bloc\_index.\*). Here \* represents the no split for the block.

If the total number of comparisons per block is more than the average then the block splits into m sub blocks. Here m is the number of partitions for the input data. So the total number of match tasks that is created after the split is

$$\frac{1}{2} \cdot m \cdot (m - 1) + m$$

k.i will be the key component denoted for the m match task. k.i\*j will the key component denoted for the  $\frac{1}{2} \cdot m \cdot (m - 1)$  match tasks here i and j belongs to [0,m-1] and  $i < j$ . The advantage of splitting into m sub blocks is that the m will be relatively growing based on the input size of the data thus a generalized way of handling larger and larger inputs sets of data.

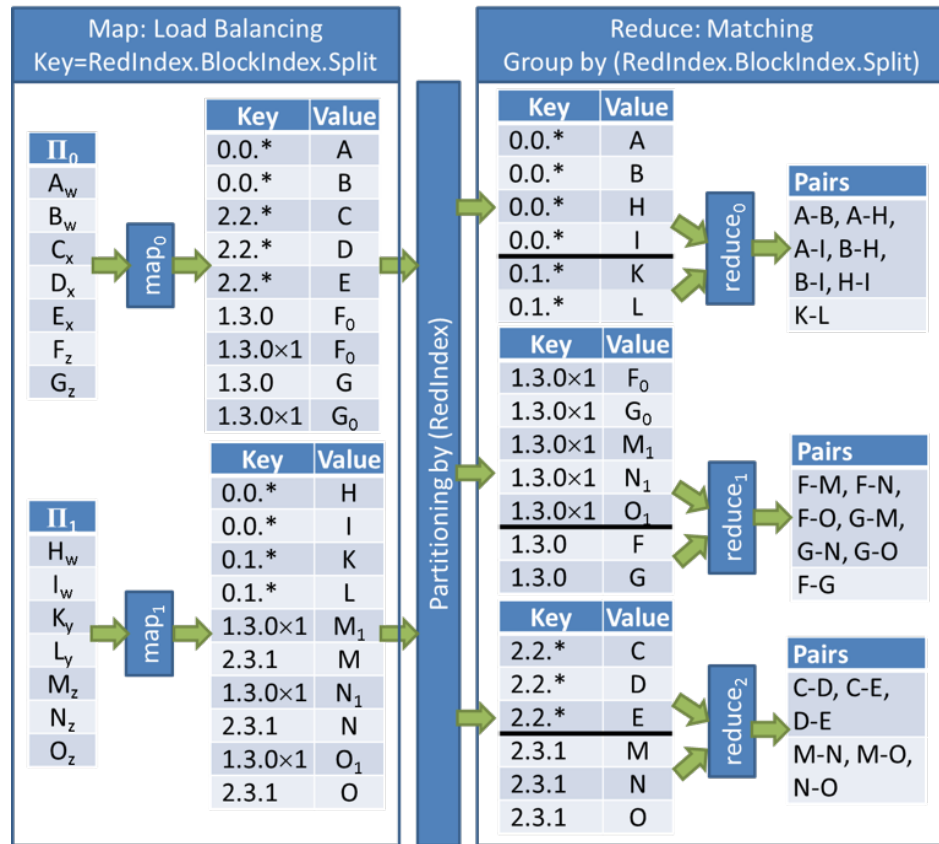
All the reduce tasks are numbers from 0 to r-1 which can be used to assign each partition to a desired reduce task. Now we group entities based on entire key to ensure that reduce task will receive all the entities within the same block, and increasing the m will also decrease the physical memory consumption, and since the block into split and m number of match task have been created this decreases the number of entity comparison per reduce block.

All the blocks tasks created in the map function is now sorted based on the number of comparison per block and then assigns to the reduce task. This way larger blocks are processed first and can be done faster as not other tasks has been assigned to the reduce task.

**BDM for the below example**

			Partition	
			$\Pi_0$	$\Pi_1$
Blocks	w	$\Phi_0$	2	2
	y	$\Phi_1$	0	2
	x	$\Phi_2$	3	0
	z	$\Phi_3$	2	3

**Figure 9 : BDM generated from the example of 14 entities**



**Figure 10 : MapReduce workflow for Block Split**

Looking into the figure above it is clear that block with blocking key z is split into 2 sub blocks, looking into the BDM for the above example states that the z.0 block has 2 and z.1 has 3 comparisons per input partition, so total after the split 3 match task has been created with keys for sub blocks as (3.0, 3.0\*1, 3.1) resulting 1,6,3 comparison.

```

map configure(m, r, partitionIndex)
{
  matchTasks ← empty map;
  compsPerReduceTask ← BDM.pairs()/r;
  // Read BDM from reduce output of Algorithm
  BDM ← readBDM();
  // Match task creation
  for k ← 0 to BDM.numBlocks()-1 do
  {
    comps ← 1/2 · BDM.size(k) · (BDM.size(k) - 1);
    if 0 < comps ≤ compsPerReduceTask then
      matchTasks.put((k, 0, 0), comps);
    else if comps > 0 then
      {
        for i ← 0 to m-1 do
          blockitok ← BDM.size(k, i);
          for j ← 0 to i do
            blockjtok ← BDM.size(k, j);
            if blockitok * blockjtok > 0 then
              if i=j then
                matchTasks.put((k, i, j), 1/2 * blockitok * (blockitok-1)
              }
            }
          else
            matchTasks.put((k, i, j), blockitok * blockjtok)
        }
      }
  }
}

```

Figure 11 : Algorithm for Block Split Map task

```

// Reduce task assignment
matchTasks.orderByValueDescending();
{
foreach ((k,i,j), comps) ∈ matchTasks) do
    {
        reduceTask ← getNextReduceTask();
        matchTasks.put((k, i, j), reduceTask);
        addCompsToReduceTask(reduceTask, comps);
    }
}

```

Figure 12 : Algorithm for Block Split Reduce task

### 4.3 Match Algorithms:

Now to specify the match for the pair of entity I have implemented the basic Edit Distance Matching algorithm, this algorithm take the two entities as input and emits the % of similarity. I used this as base, if the match is more than the threshold, I declare it as match and if it is less than the threshold it is consider as non-match.

### Edit Distance Algorithm Implemented in Java

```
public static double similarity(String s1, String s2) {
    String longer = s1, shorter = s2;
    if (s1.length() < s2.length()) {
        // longer should always have greater length
        longer = s2; shorter = s1;
    }
    int longerLength = longer.length();
    if (longerLength == 0) { return 1.0; /* both strings are zero length */ }
    return (longerLength - editDistance(longer, shorter)) / (double) longerLength;
}

public static int editDistance(String s1, String s2) {
    s1 = s1.toLowerCase();
    s2 = s2.toLowerCase();

    int[] costs = new int[s2.length() + 1];
    for (int i = 0; i <= s1.length(); i++) {
        int lastValue = i;
        for (int j = 0; j <= s2.length(); j++) {
            if (i == 0)
                costs[j] = j;
            else {
                if (j > 0) {
                    int newValue = costs[j - 1];
                    if (s1.charAt(i - 1) != s2.charAt(j - 1))
                        newValue = Math.min(Math.min(newValue, lastValue),
                            costs[j]) + 1;
                    costs[j - 1] = lastValue;
                    lastValue = newValue;
                }
            }
        }
        if (i > 0)
            costs[s2.length()] = lastValue;
    }
    return costs[s2.length()];
}
```

Figure 13 : Edit distance algorithm implemented in Java

## CHAPTER 5

### Implementation

#### 5.1 Spark Cluster Setup:

I had done experiment in two modes one VM mode and another local cluster mode. In VM mode I have setup Ubuntu 14 cluster with 3 nodes 2 GB of RAM and 80GB of virtual storage for each node

1. Install Java SDK on to the VM

```
1  # optional - remove openjdk if your installed it
2  sudo apt-get purge openjdk*
3  # install Oracle Java SDK 6
4  sudo add-apt-repository ppa:webupd8team/java
5  sudo apt-get update
6  sudo apt-get install oracle-java6-installer
7  # specify the JAVA_HOME environment variable in /etc/environment
8  sudo nano /etc/environment
9  JAVA_HOME=/usr/lib/jvm/java-6-oracle/
10 # force OS to reload the /etc/environment file
11 source /etc/environment
```

Figure 14 : Set of Instructions to install JAVA SDK on to VM

2. Install Scale and verify the installation using the command scale -version
3. Provide remote accesses to all the VM inside the cluster, I have used SSH

```
1  # On Worker nodes, we install SSH Server so that we can access this node from Master
2  sudo apt-get install openssh-server
3  # On Master node, we generate a rsa key for remote access
4  ssh-keygen
5  # To access Worker nodes via SSH without providing password (just use our rsa key),
6  ssh-copy-id -i ~/.ssh/id_rsa.pub <username_on_remote_machine>@<IP_address_of_that_node>
7  # Example:
8  ssh-copy-id -i ~/.ssh/id_rsa.pub ntkhoa@192.168.85.136
```

Figure 15 : Set of instructions to provide remote access to all VMs

keygen



4. Install HDFS file system-using hadoop and configure hadoop library.

```
1  # Config Hadoop environment variables
2  nano ~/.bashrc
3  # add following lines
4  #HADOOP VARIABLES START
5  export JAVA_HOME=/usr/lib/jvm/java-6-oracle/
6  export HADOOP_INSTALL=/usr/local/hadoop
7  export PATH=$PATH:$HADOOP_INSTALL/bin
8  export PATH=$PATH:$HADOOP_INSTALL/sbin
9  export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
10 export HADOOP_COMMON_HOME=$HADOOP_INSTALL
11 export HADOOP_HDFS_HOME=$HADOOP_INSTALL
12 export YARN_HOME=$HADOOP_INSTALL
13 export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
14 export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib/native"
15 #HADOOP VARIABLES END
16 source ~/.bashrc
```

Figure 16 : Set of Instruction to configure hadoop file system

5.

- Config settings:

***/etc/hadoop/core-site.xml***

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9000</value>
</property>
```

***etc/hadoop/yarn-site.xml***

```
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

***etc/hadoop/mapred-site.xml***

```
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
```

***etc/hadoop/hdfs-site.xml***

```
mkdir -p ~/hadoop_store/hdfs/namenode
mkdir -p ~/hadoop_store/hdfs/datanode
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:/home/ntkhoa/hadoop_store/hdfs/namenode</value> <!-- Path to store NameNode
data in your local folder-->
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/home/ntkhoa/hadoop_store/hdfs/datanode</value> <!-- Path to store NameNode
data in your local folder-->
</property>
```

## 6. Download spark and install.

- You can download [Spark 1.2.1 prebuilt for Hadoop 2.4 here](#).

The latest release of Spark is Spark 1.2.1, released on February 9, 2015 ([release notes](#)) ([git tag](#))

1. Choose a Spark release:
2. Choose a package type:
3. Choose a download type:
4. Download Spark: [spark-1.2.1-bin-hadoop2.4.tgz](#)
5. Verify this release using the [1.2.1 signatures and checksums](#).

- Save and extract that under your home folder.

\* Do some configurations:

- Specifies the Worker Nodes for the Master Node by:

+making a copy of the file `./conf/slaves.template` and name it `./conf/slaves` at Master Node, then remove the "localhost" line, and add the IP addresses of your Worker Nodes line by line.

- Similarly, change some configuration of the file `./conf/spark-env.sh`:

+ export SPARK\_MASTER\_IP=192.168.85.135 # the IP address of the Master Node so that the Worker

Nodes know where to connect to

+ export SPARK\_WORKER\_CORES=1

+ export SPARK\_WORKER\_MEMORY=800m

+ export SPARK\_WORKER\_INSTANCES=2

# PS: mine, the Master Node has IP 192.168.85.135 and the Worker Node has IP 192.168.85.136

**Figure 17: Set of Instructions to Download and Install Spark on VM cluster**

7. Start the Spark Cluster:

```
1 | ./sbin/start-all.sh # start our cluster
2 | ./sbin/stop-all.sh # if you want to stop our cluster
```

8. Open any browser and validate the Spark UI context by typing Master IP :8080 ports

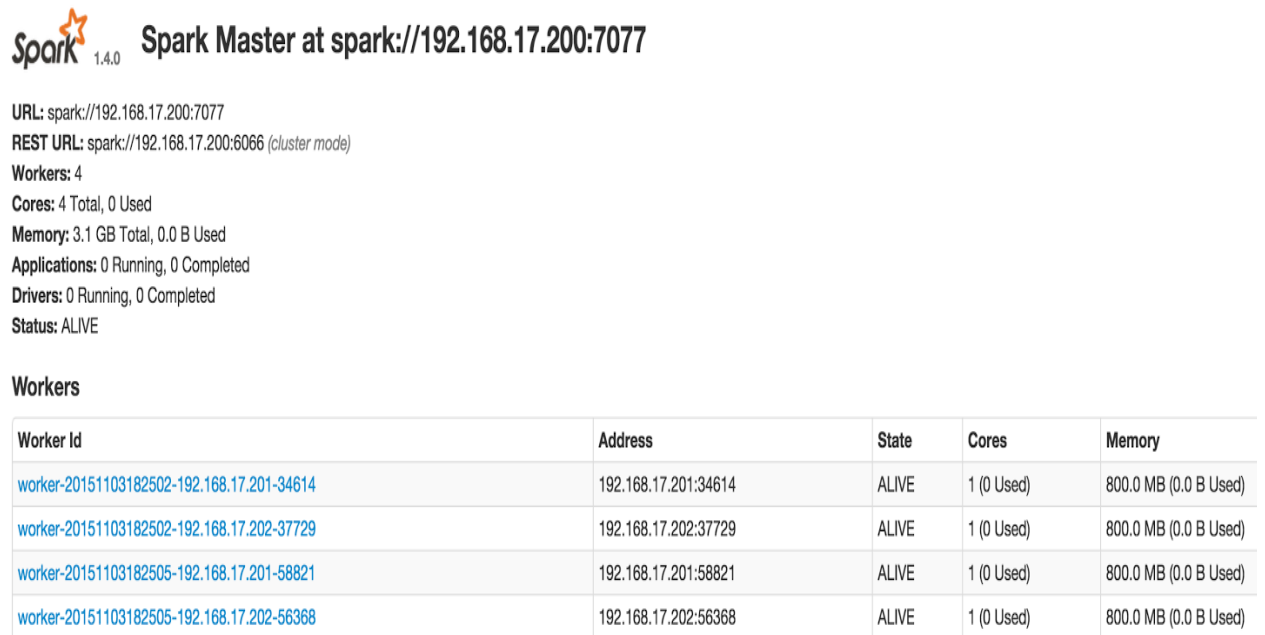


Figure 18: Spark Web UI context with nodes information

## 5.2 Installations and Configuration

### Installations

- Apache Spark 1.5.0 with Hadoop 2.6
- Java 1.8
- AWS CLI

### Configurations

#### Standalone

My Machine has Intel i7 Processor, 16 Gigabytes RAM, Master: local [4] ( 4 cores)

```
SparkConf().setMaster("local[4]").setAppName("MyApp");
```

### **Virtual Machine Cluster**

I have setup Ubuntu 14 cluster with 3 nodes 2 GB of RAM and 80GB of virtual storage for each node

```
SparkConf().setMaster("spark://192.168.92.87:7077").setAppName("MyApp");
```

### **Amazon AWS EMR Cluster with S3 Storage**

-EMR 4.1.0 with Apache Spark 1.5.0 with Hadoop 2.6

-Master: 1, Slaves: 2

- EC2 instance (m3.xlarge) with 4 CPU Cores and 16 GB RAM each

## **5.2 Sample Code**

**Creating spark context and setting the jars files to run in cluster, and reading the input data**

```
String logFile1 = "/home/akondra/Desktop/iad.csv";

SparkConf conf = new SparkConf().setMaster("spark://192.168.17.194:7077").setAppName("MyApp");

String jars[]={"/home/akondra/Desktop/TestSpark/target/TestSpark-0.0.1-SNAPSHOT.jar"};

conf.setJars(jars);

JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> logdata1 = sc.textFile(logFile1).cache();
```

**Figure 19 : Spark Code to create spark context**

## Sample Code to Implement BDM

```
JavaRDD<String> lines1 = logdata1.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) {
        return Arrays.asList(s.split("\n"));
    }
});

JavaPairRDD<String, String> wordToCountMap1 = lines1.flatMapToPair(new PairFlatMapFunction<String, String, String>() {
    public List<Tuple2<String, String>> call(String s) throws Exception {
        List<Tuple2<String, String>> results = new ArrayList<Tuple2<String, String>>();
        TaskContext task = TaskContext.get();
        System.out.println("task Partition id"+task.partitionId());
        String key[] = s.split(",");
        for(int i=1;i<key.length;i++)
        {
            if(!key[i].isEmpty())
            {
                results.add(new Tuple2<String, String>(key[i] + "." + task.partitionId(), "1"));
                results.add(new Tuple2<String, String>("Ignore_" + key[0], "Ignore_" + key[i]));
            }
        }
        return results;
    }
});

JavaPairRDD<String, String> wordCounts1 = wordToCountMap1
    .reduceByKey(new Function2<String, String, String>() {
        public String call(String first, String second) throws Exception {
            if(first.contains("Ignore_")||second.contains("Ignore_"))
            {
                return first + "," + second;
            }
            else
            {
                int f1 = Integer.parseInt(first);
                int s1 = Integer.parseInt(second);
                int sum = f1+s1;
                String rsum=Integer.toString(sum);
                return rsum;
            }
        }
    });
```

Figure 20 : Spark code to implement BDM

## Sample Block Distribution matrix

\*\*\*\*\*  
 \* Block Distribution matrix\*  
 \*\*\*\*\*

Key	0	1	combinations
Software Engineering	6	3	36
India	6	4	45
USA	3	5	28
Computer Science	3	6	36

TOTAL COMPARISON POSSIBLE = 145

## Sample code of map function to create block splits and generate key values pairs accordingly

```
JavaPairRDD<String, String> wordToCountMap3 = lines3
    .flatMapToPair(new PairFlatMapFunction<String, String, String>() {

        public List<Tuple2<String, String>> call(String s) {
            List<Tuple2<String, String>> results = new ArrayList<Tuple2<String, String>>();
            String key[] = s.split(",");
            for (int i = 1; i < key.length; i++) {
                if (!key[i].isEmpty()) {
                    System.out.println("Key is " + key[i]);

                    int blkindex = entitykey.get(key[i]);
                    System.out.println("Block index" + blkindex);
                    int empcp = entitykeycomp.get(key[i]);
                    TaskContext task = TaskContext.get();
                    if (percentagecal > empcp) {
                        results.add(new Tuple2<String, String>("0." + blkindex + ".*", key[0] + "{" +
                            culusterSignature.get("Ignore_" + key[0]) + "}"));
                    } else {

                        results.add(new Tuple2<String, String>(task.partitionId() + "." + blkindex + "." +
                            0, key[0] + "{" + culusterSignature.get("Ignore_" + key[0]) + "}"));
                        String x = key[0] + "hiphen" + task.partitionId();
                        results.add(new Tuple2<String, String>("S." + blkindex + ".0X1", x + "{" +
                            culusterSignature.get("Ignore_" + key[0]) + "}"));
                    }
                }
            }

            return results;
        }
    });

wordToCountMap3.groupByKey();
```

Figure 21 : Spark code to implement map and reduce function

### Sample Output of Entity Matching after block split

```
[Jhon Ham,Kinley P] has 0.0 Similarity
[Jhon Ham,C Bing] has 0.0 Similarity
[Jhon Ham,Bing] has 0.125 Similarity
[Jhon,Petter] has 0.0 Similarity
[Jhon,Kinley P] has 0.0 Similarity
[Jhon,C Bing] has 0.16666666666666666 Similarity
[Jhon,Bing] has 0.0 Similarity
[Petter,Bing] has 0.0 Similarity
[Kinley P,C Bing] has 0.0 Similarity
[Dexter Morgen,Chris Pollet] has 0.23076923076923078 Similarity
=====
[C Bing,Bing] has 0.6666666666666666 Similarity
=====

=====
[Dan McGee,Dan McGee] has 1.0 Similarity
=====
[Dexter Morgen,Morgen D] has 0.3076923076923077 Similarity
[Dexter Morgen,P Chris] has 0.15384615384615385 Similarity
[Dexter Morgen,Dan McGee] has 0.38461538461538464 Similarity
[Dexter Morgen,Dan McGee] has 0.38461538461538464 Similarity
[Morgen D,Chris Pollet] has 0.08333333333333333 Similarity
[Chris Pollet,P Chris] has 0.25 Similarity
[Chris Pollet,Dan McGee] has 0.16666666666666666 Similarity
[Chris Pollet,Dan McGee] has 0.16666666666666666 Similarity
[Morgen D,Dexter] has 0.125 Similarity
[Morgen D,Dan McGee] has 0.0 Similarity
[H Jhon,Petter Kinley] has 0.07692307692307693 Similarity
[H Jhon,Changluar Bing] has 0.14285714285714285 Similarity
[H Jhon,Dexter] has 0.0 Similarity
[H Jhon,Chirstopher A] has 0.15384615384615385 Similarity
[H Jhon,Dan McGee] has 0.11111111111111111 Similarity
[Petter Kinley,Dexter] has 0.3076923076923077 Similarity
[Petter Kinley,Chirstopher A] has 0.0 Similarity
[Changluar Bing,Dexter] has 0.07142857142857142 Similarity
[Changluar Bing,Chirstopher A] has 0.14285714285714285 Similarity
[Dexter,Chirstopher A] has 0.23076923076923078 Similarity
[Dexter,Dan McGee] has 0.22222222222222222 Similarity
[Chirstopher A,Dan McGee] has 0.07692307692307693 Similarity
[Jhon Ham,H Jhon] has 0.25 Similarity
[Jhon Ham,Chirstopher A] has 0.15384615384615385 Similarity
=====
[H Jhon,Jhon] has 0.6666666666666666 Similarity
=====
```

Figure 22 : Sample Output of Entity Matching after Block Split

The output has the highlighted area with “==” represents sample entity pairs whose similarity index is more than 0.5

### 5.3 Execution Report

I had done experiment in two modes one VM mode and another local cluster mode.

In VM mode I have setup Ubuntu 14 cluster with 3 nodes 2 GB of RAM and 80GB of virtual storage for each node

```
SparkConf().setMaster("spark://192.168.92.87:7077").setAppName("My App");
```

in local mode I have use Local mode with 4 nodes

```
SparkConf().setMaster("local[4]").setAppName("MyApp");
```

Spark distribution:

To perform EM, I have used the same data set as source and comparison index; I have depended on the default partition index of the spark, which divides the data to be of default partition size of 64 mb

```
String logFile1="G://Testing.txt";
```

```
JavaRDD<String> logdata1= sc.textFile(logFile1,2).cache();
```

Report runtime performance (time measured for different steps in the process), i.e. “what you measured”

**Data Loading:** How long did it take to load the data? : I initially cache the data into a RDD

```
JavaRDD<String> logdata1= sc.textFile(logFile1,2).cache();
```

Later the same RDD is given to the algorithm twice, once for calculating the BDM and other entity comparisons

For every job

Below all time comparison is based on more than 3,89,000 entities



**Time for job initiation:** Load data time of execution is 287 seconds

**Time for map task**

**First Map Task** around 780 seconds for map task for BDM

**Second Map Task** 1027 seconds for map Task for data comparisons and block splitting

**Time for reduce task:**

**First Reduce Task:** count number of comparison and compute BDM 970 seconds

**Second Reduce Task:** Generating pairs and ignore few comparison pairs 1378 seconds minutes because the of blocking, similar keys are mapped in same node and reduced in same node where it was mapped

**Time for writing data back to disk:** In my case Spark had 2 times writing the data into disk, one for sending calculated BDM into Disk and another for Pairs after all comparisons, which is maximum time,

**Workload distribution among nodes:** Spark Default partition and workload distributions among 4 nodes

## **CHAPTER 6**

### **Performance Evaluation**

I have Evaluated Block Split in four critical factors

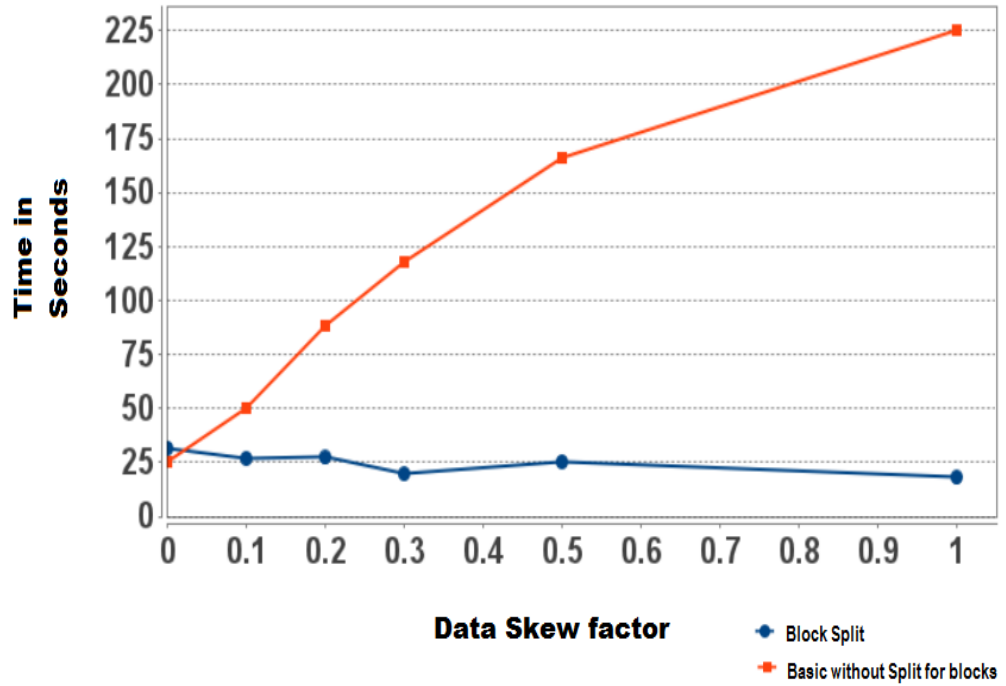
1. Degree of Data Skew
2. Block Split Blocking in terms of Precision and Recall
3. Configuring number of maps and reduce tasks and Number of available nodes in clusters

#### **6.1 Degree of Data Skew**

To validate the robustness of our block split load-balancing strategies against data skew .I generated different sizes of blocks by modifying the blocking function. The number of entity pairs depends up on the distribution of entities over all blocks.

I calculated the average execution time for different data skews.

1. Block Split when comparing with Basic load balancing strategy, it was slow for a uniform block distribution because it is additionally calculating BDM .So when the block size of all the blocks are close to uniform distribution the overhead of calculating the BDM increased the execution time.
2. But in other case where the block distribution of entities was not uniform and as we increase the data skew the effect of calculating the BDM became invisible, and because of the splits into sub blocks, the imbalance caused with reduce task is handled and the execution time for generating pairs become stable.



**Figure 23 : Evaluation report for Block Split execution with different data skew factors**

In the above figure Data Skew factor 0 means uniform distribution entities and 1 means all the entities are inside on block

## 6.2 Block Split blocking evaluation in terms of Precision and Recall:

**Recall:** It is the ratio of number of relevant entities that are found in the data source to total number all relevant entities in the data source.

**Precision:** It is the ratio of number of relevant entities that are found in the data source to total number all entities in the data source.

The basic strategy of Block split assumes a standard blocking techniques where blocks are disjoint and processed separately, i.e. each record is assigned to a single block due to this reason the ratio for recall is very less as it only matches one record only once for one block.

For example consider below records and Employee Name is the entity that needs to be matched again all other entities

	<b>Employee Name</b>	<b>Employee Location</b>	<b>Employee Department</b>	<b>Source Of Info</b>
1.	Jhon Ham	India	Computer Science	Facebook
2.	Jhon H	USA	Computer Science	Google+
3.	H Jhon	India	Science	Facebook

When I use blocking key as “Employee Location”. For example “India”, I got [1,3] records into one block. When I used “Employee Department” I got [1,2] records into one block, If I want to run entity matching of each keys or may be combination of fields into keys block split limits to use only one blocking key.

### 6.2.1 Optimization

To improve recall we can apply multiple clustering with different blocking keys iteratively. Clusters of different blocking keys can be created simultaneously when reading the records in the map phase. In the example, Employee Location would give two clusters, one for India and one for USA records.

Blocking key department would also give two clusters for Department (CS, Science). Every record can be assigned to two clusters in the map phase and has then to be redistributed twice to the reducers handling the respective clusters, that is the record 1 goes to the India and CS clusters and has to be sent to the two reducers handling these clusters. With block split we can analyze the sizes of all clusters in a separate analysis MR job (BDM) that also determines necessary splits and the redistribution function.

But, with multiple blocking keys we will likely find more matches thus increase recall but we may have many redundant matches for pairs of records that are in more than

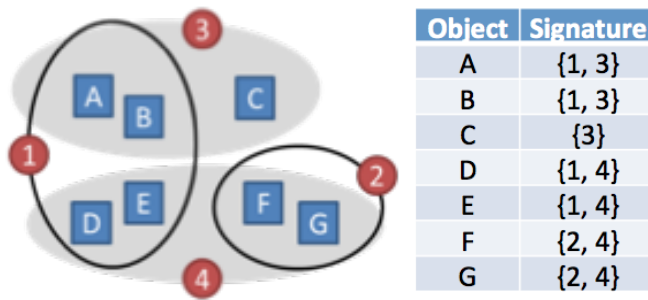
one cluster. These redundant matches can be easily avoided, below mentioned optimization to avoid matching the same pairs in different clusters helps the reducers to save unnecessary match work .

### 6.3 Signature Based Entity Comparison:

This is implemented with the Match job of block split, Our proposed optimization does not need any extra map or reduce task but with in the Match job's map and reduce task we implement this signature base entity comparison

#### Signature function:

This function determines all the blocking clusters into which this records goes into, for example consider the below image



Entity A is present in cluster 1 and 3 and entity F is present in 2 and 4 similarly for all the records

**Map Job:** for every record, the map function determines all its signatures, sorts all the signatures in the accessing order and associated to the value of the key value pair generated from the map function.

**[key, value] = [reduceindex.blockingkey.Split , Entity ,{Signatures}]**

Example: [0.1. \*, Jhon H, {1,3,4}]

**Reduce:** The reduce phase will now introduce another check; it selects the least common signatures for pair of entities and apply comparison for the least value of signature and ignore all other signature during the comparison.

Example:

1. Entity Pair [ **Jhon H {1,3,4}** , **Jhon {1,3,4}** ] during reduce and group by keys when blocking key is 0.1.\*
2. [ **Jhon H** , **Jhon** ] is pair is generated and when blocking key is 0.3.\* and 1.4.\* the pair generation is ignored.

```
map(kin=unused, vin=o)
  S ←  $\sigma(o)$ .distinct();
  S.sort();
  SS ← [] // smaller signature list
  foreach si ∈ S do
    output(ktmp = si, vtmp= (o, SS));
    SS.append(si);

reduce(ktmp=s, list(vtmp)=list(object, SS))
  buf ← {};
  foreach (o1, SS1) ∈ list(object, SS) do
    foreach (o2, SS2) ∈ buf do
      if doOverlap (SS1, SS2) then
        compare(o1, o2);
  buf ← buf ∪ {(o1, SS1)};
```

Figure 24 : Algorithm to implement Signature Based Entity Matching

### Output after Implementation:

To illustrate the working of Signature base entity comparison using block split let the consider example show to the below.

- Imagine if we only considered Blocking key 1 and generated pairs we might lose [ C-G ] and [D-I] [B-H] pairs for comparison as can be grouped into a block only with blocking key 2
- Similarly if we had choose Blocking key 2 and generated pairs we might lose [A-C] [A-D] [B-C] [B-D] [G-H] [H-I] etc pair for comparison as they can be grouped only using blocking key 1
- If we generate blockings using bot blocking key 1 and blocking key 2 Pairs [A-B] [C-D] would had compared twice as these pairs are present in both the blocking

keys

Entities	Blocking key 1	Blocking key 2
A	1	3
B	1	3
C	1	4
D	1	4
G	2	4
H	2	3
I	2	4

Figure 25 : Set of entities to multiple key blocking

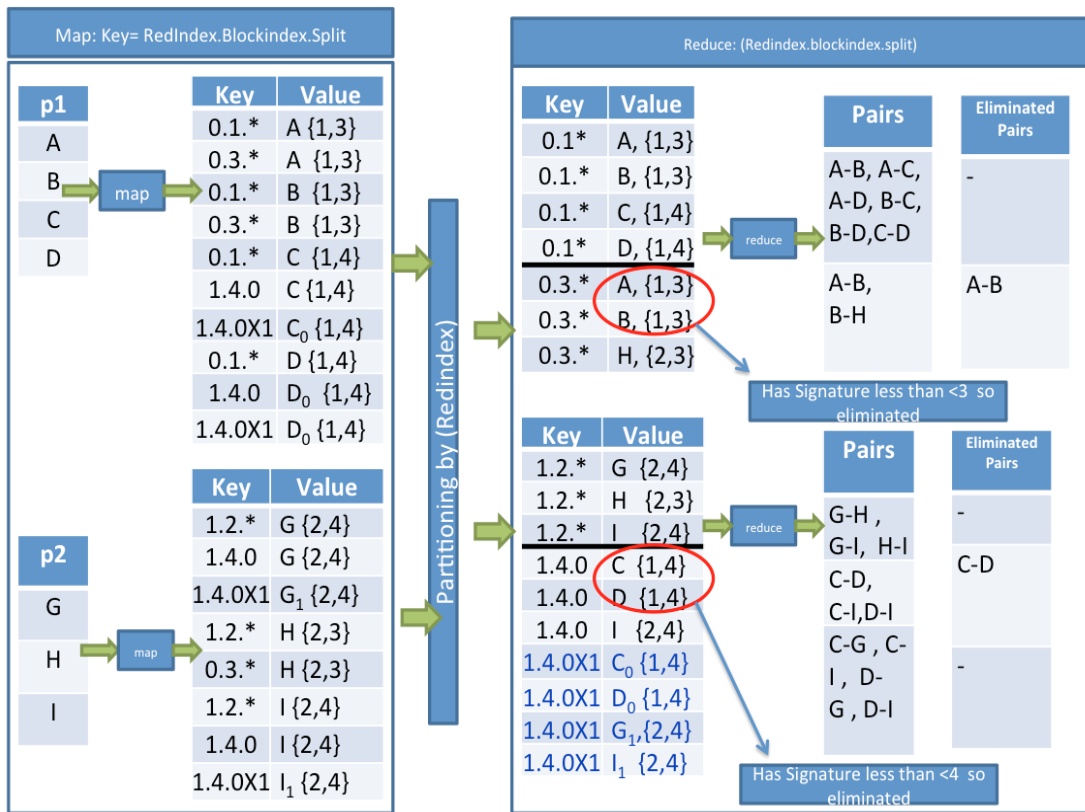
## Sample Output after Signature Based Entity Matching using Block Split

```

1.3.0: [Dexter Morgen{1/3},Morgen D{2/3},Dexter{2/3},Chris Pollet{1/3},P Chris{1/3},Christopher A{2/3}]
[Dexter Morgen,Morgen D] has 0.3076923076923077 Similarity
[Dexter Morgen,Dexter] has 0.46153846153846156 Similarity
Eliminated Pair ==> [Dexter Morgen,Chris Pollet]
Eliminated Pair ==> [Dexter Morgen,P Chris]
[Dexter Morgen,Christopher A] has 0.07692307692307693 Similarity
Eliminated Pair ==> [Morgen D,Dexter]
[Morgen D,Chris Pollet] has 0.08333333333333333 Similarity
[Morgen D,P Chris] has 0.0 Similarity
Eliminated Pair ==> [Morgen D,Christopher A]
[Dexter,Chris Pollet] has 0.08333333333333333 Similarity
[Dexter,P Chris] has 0.0 Similarity
Eliminated Pair ==> [Dexter,Christopher A]
Eliminated Pair ==> [Chris Pollet,P Chris]
[Chris Pollet,Christopher A] has 0.3076923076923077 Similarity
[P Chris,Christopher A] has 0.15384615384615385 Similarity
0.2.*: [H Jhon{2/3},Petter Kinley{0/2},Changluar Bing{0/2},Morgen D{2/3},Dexter{2/3},Christopher A{2/3},Dan McGee{0/0/1/2},McGee Dan{0/2}]
[H Jhon,Petter Kinley] has 0.07692307692307693 Similarity
[H Jhon,Changluar Bing] has 0.14285714285714285 Similarity
[H Jhon,Morgen D] has 0.125 Similarity
[H Jhon,Dexter] has 0.0 Similarity
[H Jhon,Christopher A] has 0.15384615384615385 Similarity
[H Jhon,Dan McGee] has 0.11111111111111111 Similarity
[H Jhon,McGee Dan] has 0.11111111111111111 Similarity
Eliminated Pair ==> [Petter Kinley,Changluar Bing]
[Petter Kinley,Morgen D] has 0.15384615384615385 Similarity
[Petter Kinley,Dexter] has 0.3076923076923077 Similarity
[Petter Kinley,Christopher A] has 0.0 Similarity
Eliminated Pair ==> [Petter Kinley,Dan McGee]
Eliminated Pair ==> [Petter Kinley,McGee Dan]
[Changluar Bing,Morgen D] has 0.14285714285714285 Similarity
[Changluar Bing,Dexter] has 0.07142857142857142 Similarity
[Changluar Bing,Christopher A] has 0.14285714285714285 Similarity
Eliminated Pair ==> [Changluar Bing,Dan McGee]
Eliminated Pair ==> [Changluar Bing,McGee Dan]

```

Figure 26 : Sample Entity Matching Output after Signature Based Entity Matching



**Figure 27 : Workflow for Signature Based Entity Matching with Block Split**

Below all time comparison is based on more than 3,89,000 entities

Time for job initiation : Load data stage 0 time of execution is 287 seconds

Time for Analysis Job (BDM and Signature) generation:

- **First Map Task** : around 1730 seconds for map task for BDM , this is due to creating blocking key for multiple entities
- **First Reduce Task** : count number of comparison and compute BDM 2170 seconds for multiple blocking keys and Signature generation

Time for Match Job (Block Split with Signature Based Entity Comparison) generation:

- **Second Map Task** : 2327 seconds for map Task for data comparisons and block splitting, and appending Signature to it
- **Second Reduce Task** : Generating pairs and performing additional check to ignore redundant comparison pairs 3087 seconds.



### **6.3 Increase in number of Reduce Tasks and nodes**

As the number of reducers increase, the execution time decreases as the distribution reduce task has increased. Block Split provides very good stable execution times because of its load balancing effectiveness. But it take the overhead of Calculating the BDM, on the other hand after mapping is done for a larger data sets the time of group and collect and then reduce is taking longer time

Issue is that the Block Split's load balancing strategy depends on the input (map) partitioning, and network traffic in reducing the tasks, Solution Observed that using a sorted input dataset is likely to group together large blocks into the same map partition. This limits Block Split's ability to split large blocks and deteriorates its execution time. The Block Split strategy shows a step-function-like behavior because the number of reduces tasks determines what blocks will be split but do not influence the split method itself, which is solely based on the input partitions. Faster entity resolution by Blocking Parallel matching

### **6.4 Advantages of Improved Block Split**

1. Faster entity resolution by Blocking Parallel matching
2. The load imbalance problem has been addressed by Block Split, a general load balancing MR-based approach that takes the size of blocks into account.
3. High Recall is achieved and a balanced the precision using signature based entity comparison.
4. Implementing using spark reduces the multiple writes into disc to a single entry to disc.

## **CHAPTER 7**

### **Conclusion**

The proposed load balancing approach Block Split for parallelizing blocking-based entity resolution using the widely available MapReduce framework is capable to deal with skewed data (blocking key) distributions and effectively distribute the workload among all reduce tasks by splitting large blocks. Our evaluation in a real cluster environment with one master and 2 worker nodes using real-world data demonstrated that the approach is robust against data skew and scale with the number of available nodes. The optimized Block Split approach using Signature Based Entity Comparison improved the Recall and stabilized the precision for minimizing redundant entities pair comparison.

## CHAPTER 8

### References

- [1] L. Kolb, A. Thor, and E. Rahm, “Block-based Load Balancing for Entity Resolution with MapReduce,” in CIKM, 2011.
- [2] R. Baxter, P. Christen, and T. Churches, “A comparison of fast blocking methods for record linkage,” in Workshop Data Cleaning, Record Linkage, and Object Consolidation, 2003
- [3] Hanna, and Erhard Rahm “Framework for Entity Matching by ”,2009  
Framework for Entity Matching by Erhard Rahm [http://dbs.uni-leipzig.de/file/FrameworksForEntityMatchingAComparison\\_dke.pdf](http://dbs.uni-leipzig.de/file/FrameworksForEntityMatchingAComparison_dke.pdf)
- [4] Load Balancing for entity matching by Erhard Rahm  
[http://dbs.uni-leipzig.de/file/ICDE12\\_conf\\_full\\_088.pdf](http://dbs.uni-leipzig.de/file/ICDE12_conf_full_088.pdf)
- [5] Redundancy-free Similarity Computation by Erhard Rahm  
[http://dbs.uni-leipzig.de/file/pair\\_wise\\_comparison\\_mr.pdf](http://dbs.uni-leipzig.de/file/pair_wise_comparison_mr.pdf)
- [6] Edit Distance Similarity function  
[https://en.wikipedia.org/wiki/Edit\\_distance](https://en.wikipedia.org/wiki/Edit_distance)
- [7] Spark Mapreduce Implementation by clourera  
<https://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/>
- [8] Signature Based Entity Resolution [Don’t Match Twice: Redundancy free Similarity Computation with MapReduce] by Erhard Rahm [http://dbs.uni-leipzig.de/file/pair\\_wise\\_comparison\\_mr.pdf](http://dbs.uni-leipzig.de/file/pair_wise_comparison_mr.pdf)