# IMPLEMENTING LOAD BALANCING FOR ENTITY RESOLUTION IN DISTRIBUTED FILE SYSTEMS USING SORTED NEIGHBOURHOOD ALGORITHMS

# ABSTRACT

Entity Matching is the task of finding entities and matching them to the same entities in the given data set. These entities can belong to a single source of data, or distributed data-sources. It takes structured data as input and process includes comparison of that structured data (entity or database record) with entities present in the knowledge database. This workflow consists of two strategies: blocking (map) and matching (reduce). One of the standard approaches to entity resolution is to use sorted neighborhoods(SN) also known as sorted reduced partitioning (SRP). However, this algorithm ignores comparison of boundary entities which is not favorable. To combat this we have implemented two strategies, ReplicationSN and JobSN by interfacing Hadoop Distributed File System and Hadoop MapReduce with the Python programming language.

# INTRODUCTION

Day by day we witness a huge demand for high speed processing of large data that is generated. But in order to process these computationally expensive tasks the power of one single system is limited.

So one can argue that we increase the power of the single system(vertical scaling).

However it turns out that the hardware is expensive. So distributed systems come into play; the concept can be simply illustrated like; instead of having one person to do a particular job, we now have many people working in coordination to achieve the same goal.

This ensures that the process is completed more quickly but involving very low expenses.

Using a distributed file system we aim to perform entity matching on clusters. Entity matching is a data-intensive task and the most economical way to handle it at large scale would be to use a distributed file system. The broad availability of MapReduce distributions such as Hadoop makes it attractive to investigate its use for the efficient parallelization of data-intensive tasks.

A distributed file system is a type of distributed system solely intended for efficient file management processing. It is mainly used to manage huge datasets which otherwise might take enormous time to process. It involves in implementation of Blocking and reducing.

A distributed file system like Hadoop performs load balancing as one of its core functionalities. For example, it can be used for process management. By balancing the load we can distribute it among the different nodes in the cluster to increase efficiency and performance.

Entity resolution (also known as object matching, de-duplication, or record linkage) is such a data-intensive and performance critical task that can likely benefit from distributed systems. Given one or more data sources, entity resolution is applied to determine all entities referring to the same real world object. It is of critical importance for data quality and data integration. One critical application of entity resolution is distributed file systems could be using it to detect file copies using file metadata instead of going and manually comparing each and every file in the cluster.

An example for entity matching:

If the user wants to search for an entity with the name "Ryan McCarthy" from the given list of entities then, entity is determined from the available list of entities.  search for all entities with the name "Ryan McCarthy" and matches it against few more measures to get the most exact result. Like ,if ryan is considered as an entity in the physical world, all the social media accounts related to Ryan are his entities in the data. Mostly done to maintain customer data on large systems.

The entity matching workflow consists of two strategies: blocking (map) and matching (reduce). Blocking strategy termed as the division of a data source into partitions or blocks done to achieve better load distribution and high performance.

The second part of the workflow consists of the strategy for matching. This aims to identify all matching entity pairs within the same partition.

Now coming to the topic at hand, entity matching. The standard approach for matching n input entities is comparison of all entities with each other, which is the Cartesian product (n * n) of all input entities.

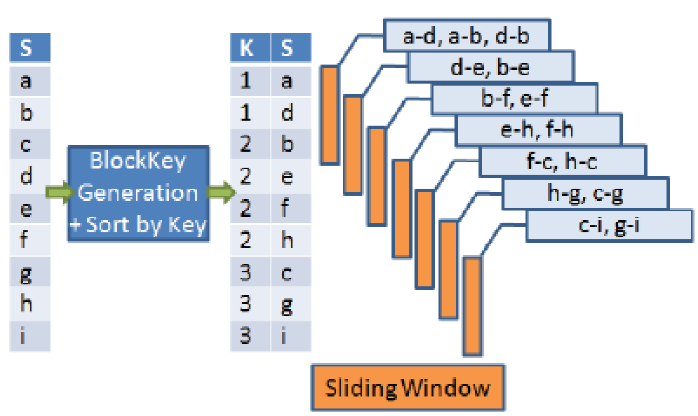
Time complexity for such approach is O(n^2).

For very large datasets, this causes intolerable execution times.

Sorted neighbourhood (SN), is where, all entities are sorted using the blocking key and then, compared with entities within a predefined specific range, which referred as a distance window w.

In this approach, the complexity of matching is O (n * w)

which is very less compared to the quadratic complexity of O(n^2).




Sorted Neighbourhood has a pitfall where it cannot detect boundary entities.

Let’s say the window size of Sorted Neighbourhood is 3, let’s say we have 2 partitioners(i.e 2 reducers), according to MapReduce, mutual access of data among partitions is not possible, so the last 2 entities in a partitions are not being compared with the first 2 entities of the next partition, These missed out entities are called boundary values. w, SN has missed

$$((r - 1) * w * (w - 1)/2)$$

boundary entity pairs.

There are two algorithms which we have implemented to solve this issue and they are called Replication Sorted Neighbourhood (RepSN) and JobSN.

In further sections we dive deep into theses algorithms.

# MODULE DESCRIPTION

Coming to the main algorithms we have implemented and their modules.

## Sorted Reduced Partitioning (SRP) / Sorted Neighbourhood (SN)

This approach is a very popular among all blocking approaches. A blocking key K, which can be the concatenation of prefixes of a few entity attributes, is determined for each of n entities. Afterwards this blocking key sorts the entities. During the next phase, a window of fixed size w then, used by sliding for comparison over all the sorted records and all entities within the range of that window compared during each slide of the window.

Total Number of Matchings or comparisons:

$(w - 1) * (n - w/2)$. W=window size n=number of entities

**Complexity analysis:**

For determining blocking key: O(n)

For sorting :(best approach) : O(nlogn)

For comparisons: O(n*w)

Total complexity: O(n) +O(nlogn) + O(n*w)

The execution of StandardSN requires a composite key. The format for which is:

<blocking key> . <partition prefix>

The partition prefix is used to dictate which partition the value belongs to and the blocking key tells us about which reducer the value is going to go to.

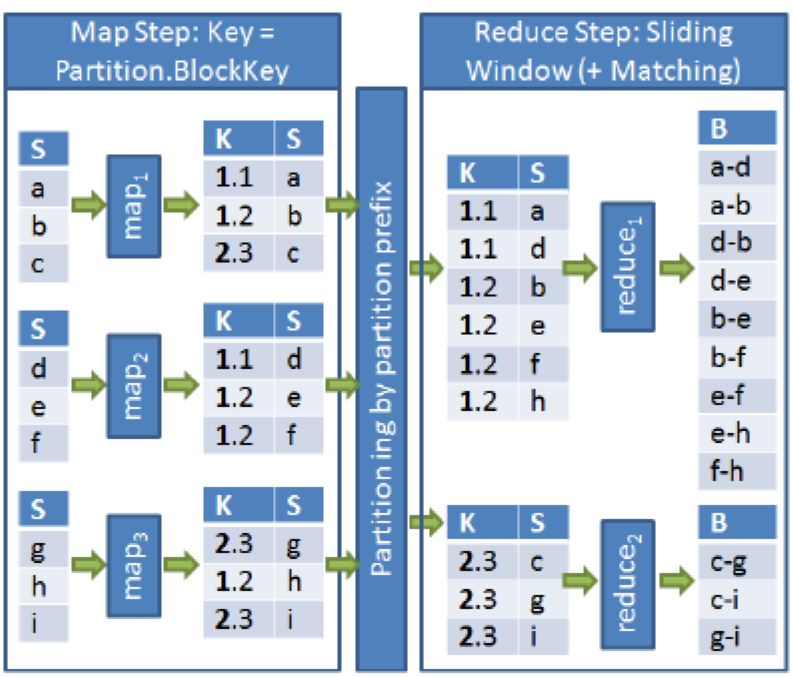



Figure 2: Example execution of sorted data partitioning

with a composite key consisting of a blocking key and a partition prefix. The pairs ( f , c), (h, c), and (h, g) can not be found since the involved entities reside in different reduce partitions.

As previously mentioned SRP doesn't take care of replicated entities. To solve this we have the RepSN and JobSN modules.

## REPLICATION SORTED NEIGHBOURHOOD (RepSN)

The RepSN approach aims to realize SN within a single MapReduce job. It extends SRP by the idea that each reduce task $i > 1$ needs to have the last $w - 1$ entities of the preceding reduce task $i - 1$ in front of its input. Instead of adding an extra map-reduce job for comparisons, RepSN will replicate the boundary entities and append to the partitions. The replica of the first w-1 entities of a partition k is appended to the last of its previous partition.

This approach make changes to the existing map function of SRP to replicate an entity that sent to both the current reducer and its successor.

The blocking key (k) and a partition prefix f (k) determines an entity key. Here, an additional boundary prefix added to differentiate between actual entities and replicated boundary entities. This comes with a trade-off for extra space.

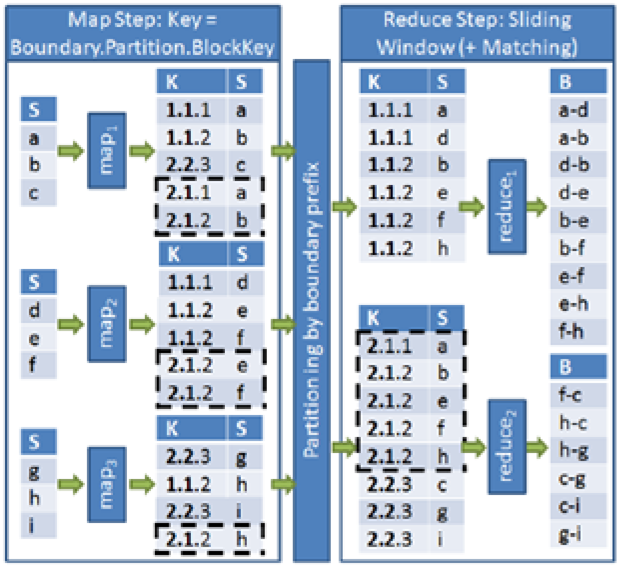



Figure 3: RepSN with window size = 3

As we can see in the above figure the boundary entities of the previous partition are replicated into the next partition. We perform this replication by first replicating the boundary entities and the modifying the blocking key of the replicated value. By changing the blocking key we have successfully directed it to the next partition.

**Algorithm 2: RepSN**

```
map_configure
    // list of the entities with the w-1 highest
    // blocking keys for each partition i<r
    foreach i ∈ {1,...,r − 1} do
        rep_i ← [];

map (key_in=unused, value_in=entity)
    k ← generate blocking key for entity;
    r_i ← p(k); // reducer to which entity is assigned by p
    bound ← r_i;
    if r_i < r then
        if sizeOf(rep_{r_i})<w-1 then
            append(rep_{r_i}, entity);
        else
            min ← determine entity from rep_{r_i} with smallest blocking key;
            k_min ← blocking key of min;
            if k >k_min then
                replace(rep_{r_i}, min, entity);
    // Use composite key to partition by bound
    output (key_tmp=bound.r_i.k, value_tmp=entity)

map_close
    foreach i ∈ {1,...,r − 1} do
        r_i ← i;
        bound ← r_i +1;
        foreach entity ∈ rep_i do
            // prefix key with r_i+1 to assign replicated
            // entities to succeeding reducer
            output (key_tmp=bound.r_i.k, value_tmp=entity)

// group by bound, order by composed key
reduce (key_tmp=bound.r_i.k, list(value_tmp)=list(entity))
    remove all entities with bound ≠ r_i from the head of list(entity) except the last w − 1;
    StandardSN (list(entity), w);
```

# JobSN Sorted Neighbourhood with additional MapReduce job

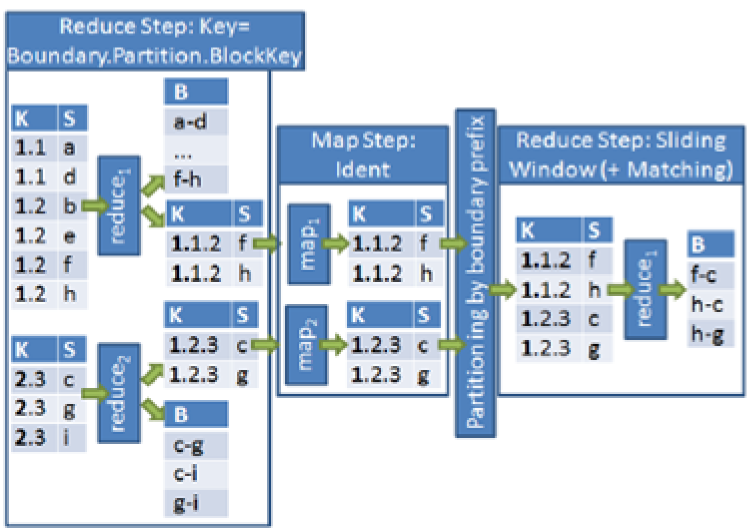
The JobSN approach utilizes SRP and employs a second MapReduce job afterwards that completes the SN result by generating the boundary correspondences. JobSN makes thereby use of the fact that MapReduce provides sorted partitions to the reduce tasks. A reduce task can therefore easily identify the first and the last w − 1 entities during the sequential execution. Those entities have counterparts in neighbouring partitions, i.e., the last w − 1 entities of a reduce task relate to the first w − 1 entities of the succeeding reduce task. In general, all reduce tasks output the first and last w − 1 entities with the exception of the first and the last reduce task. The first (last) reduce task only returns the last (first) w − 1 entities.

JobSN realizes the assignment of related boundary elements with an additional boundary prefix that specifies the boundary number. Since the last w − 1 entities of reduce task i < r refer to the ith boundary, the keys of the last w − 1 entities are prefixed with i. On the other hand, the

first w − 1 entities of the succeeding reduce task i + 1 also relate to the ith boundary. Therefore the keys of the first w − 1 entities of reduce task i > 1 are prefixed with i − 1.

The second MapReduce job of JobSN is straightforward. The map functions leaves the input data unchanged. The map output is then redistributed to the reduce tasks based

on the boundary prefix. The reduce function then applies the sliding window but filters correspondences that have already been determined in the first MapReduce job.



**Algorithm 1: JobSN**

```
// --- Phase 1 ---
map (key_in=unused, value_in=entity)
    k ← generate blocking key for entity;
    r_i ← p(k) ; // reducer to which entity is assigned by p
    // Use composite key to partition by r_i
    output (key_tmp=r_i.k, value_tmp=entity)

// group by r_i, order by composed key
reduce (key_tmp=r_i.k, list(value_tmp)=list(entity))
    StandardSN (list(entity), w) ;
    first ← first w − 1 entities of list(entity);
    last ← last w − 1 entities of list(entity);
    if r_i > 1 then
        bound ← r_i-1;
        foreach entity ∈ first do
            output (key_out=bound.r_i.k, value_out=entity)
    if r_i < r then
        bound ← r_i;
        foreach entity ∈ last do
            output (key_out=bound.r_i.k, value_out=entity)

// --- Phase 2 ---
map (key_in=bound.r_i.k, value_in=entity)
    // Use composite key to partition by bound
    output (key_tmp=bound.r_i.k, value_tmp=entity)

// group by bound, order by composed key
reduce (key_tmp=bound.r_i.k, list(value_tmp)=list(entity))
    StandardSN (list(entity), w) ;
```

Blocking may lead to partitions of largely varying size due to skewed key values. There-

fore the execution time may be dominated by a single or a few reduce tasks similar to skew effects during parallel join processing. The proposed solution is an extension of JobSN. The partitioner checks if there exists a partition with at least 3 times more than average partition size before running reduce operations over map output. If such partition exists then, we repartition that block in equal sized block equal to average partition size to balance the load across all nodes.

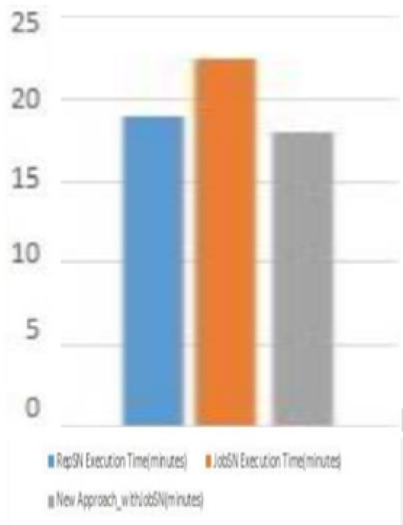
```
// — Phase 1 —
map (key=unused, value=entity)
k ← generate blocking key for entity;
ri ← p (k); // reducer to which entity is assigned by p
// Use composite key to partition by ri
output (key=ri.k, value=entity)

// group by ri, order by composed key

Foreach partition ∈ output do
If (partition_size > 3 * (total number of records / number of partitions))
        Numpartitions = partition_size / (total number of records /
number of partitions)
        Repartition (partition_number, numpartitions)

reduce (key=ri.k, list (value)=list(entity))
StandardSN (list (entity), w);
first ← first w - 1 entities of list (entity);
last w - 1 entities of list (entity);
if ri > 1 then
    bound ← ri-1;
    foreach entity ∈ first do
            output (key=bound.ri.k, value=entity)

if ri < r then
    bound ← ri;
    foreach entity ∈ last do
            output (key=bound.ri.k, value=entity)


// — Phase 2 —
map (key=bound.ri.k, value=entity)
// Use composite key to partition by bound
output (key=bound.ri.k, value=entity)
// group by bound, order by composed key
reduce (key=bound.ri.k, list(value)=list(entity))
StandardSN (list (entity), w);
```

If we take an example of 50000 entities and after map phase, there are 4 partitions with sizes 10000, 35000, 5000 and 10000 entities.

In this case, we will figure out partition 2 has most of the data and that need to be redistributed in 3 more blocks of size 12500, 12500 and 10000 and adding the boundary entities to the newly added partitions.

Once done, reducers run in parallel, which are designated matching jobs.

**Performance comparison**




Time in mins(y-axis)

# DATASET

| Task/source files | Domain | Attributes | #entities | #matches |
|---|---|---|---|---|
| DBLP-ACM | Bibliographic | title, authors, venue, year | 2614+2294 | 2224 |

# SOFTWARE USED

1. Apache Hadoop MapReduce Framework and the Hadoop Distributed File System (HDFS)

MapReduce is a programming model introduced by Google in 2004. It supports parallel data-intensive computing in cluster environments with up to thousands of nodes. A

MapReduce program relies on data partitioning and redistribution. Entities are represented by (key, value) pairs. The broad availability of MapReduce distributions such as Hadoop makes it attractive to investigate its use for the efficient parallelization of data-intensive tasks.

HDFS is the primary holder of the knowledge base upon which the data processing layer will perform search and computation processes of load balancing.

MapReduce is used to manage the block, load balancing and map phases of all the algorithms.

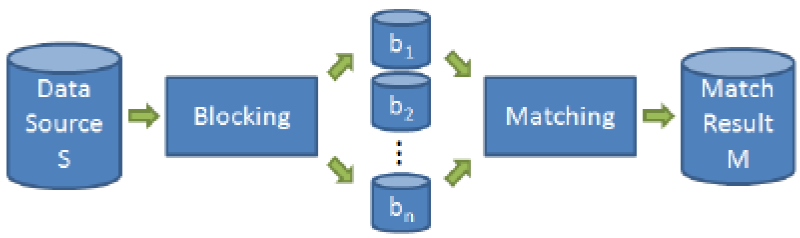



**Figure 5: MapReduce Workflow**

2. Python3 and Hadoop Streaming API

Even though Hadoop is written in Java we need not use it compulsorily. Using the Hadoop Streaming library included within the Hadoop installation we can interface HDFS and MapReduce using Python. This simplifies the coding process since Java is a strictly-typed language.The hadoop S

**Method:**

**For REPSN:**

**1.**Pass the dataset to the map-reduce job:

**2.**In Mapper phase, we set the key to be partitionPrefix+firstnames of all the authors of a paper+year of publication

**3.In Mapper we isolate the boundary entities and replicate and append them to a partition,**

**4.**In the reducer part, we print the id's of both matching papers

**5.In StandardSN we set the window length to 5, then we slide the window across the partition and do comparisons based on the authors.If two papers have atleast one same author, they are considered to be a match.(we can change the criteria however we want)**

**For JOBSN and Modified JobSN:**

**1.**Pass the data to the first phase of mapreduce.

**2.The partition prefix will generate a composite key for the same blocking key pattern that is considered in repsn.Here we consider two reducers.**

3.The mapper then will pass key,val pair to reducer.

4.**The reducer then will execute StandardSN and print the possible comparisons.**

**5.The missed comparisons for the boundary entities will also get printed as it is with their keyvalues.**

**6.Now the phase 2 mapreduce will get output from phase1 and will generate comparisons for boundary entities.**

**7.In modified job sn we just include a new partitioner which will repartition according to the algorithm discussed in the previous sections.**

## CODE

**RepSN Mapper**

```
#!/usr/bin/env python3

import sys

import re




def partition_prefix(key):

    partition_key = ""

    arr1 = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']

    arr2 = ['N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

```
if key[0] in arr1:

partition_key += '1'

if key[0] in arr2:

partition_key += '2'

# if key[0] in arr3:

#           partition_key += '3'

# if key[0] in arr4:

#           partition_key += '4'

return str(partition_key)



def map_configure(np, ws):

# creating an empty list to store the entities with the w-1 highest blocking keys for each partition i
< r

# dimensions of x = (np-1 * ws-1)

x = [[None for _ in range(ws - 1)]

for _ in range(np - 1)]

return x



def get_minimum_key(list_of_keys):

min_key = list_of_keys[0]

# sample key in list_of_keys = 1.AndrewJim-1900

for key in list_of_keys:

min_key_split = re.split('[.]', min_key)
```

```
key_split = re.split('[.]', key)
if (min(min_key_split[1].upper(), key_split[1].upper()) == key_split[1].upper()):
min_key = key
return min_key, list_of_keys.index(min_key)



def generate_blocking_key(value):
arr = ['"', '&', '#']
value = value.strip()
entity = value.split(",")
year = entity[5]
author = []
if (len(entity[3]) <= 1):
author = list("Anonymous")
else:
author = entity[3].split(';')
key = ""
for j in author:
k = j.strip()
if k[0] in arr:
# print("quote mark being printed")
k = k[1:]
elif k[-1] == '"':
# print("quote mark being printed")
```

```
        k = k[:len(k)-1]

        k = k.split(" ")

    # concatenation of authors' first names

        key = key+k[0]

        if key[0].islower():

        # print("wah wah")

        key = key.upper()

        author = [None]

        mod_key = key+"-"+str(year)

        temp_key = partition_prefix(mod_key)+"."+mod_key

        # print("mod_key: " + mod_key + "\ntemp_key: " + temp_key)

        return temp_key



def map(value):

        entity = value

        # key is the blocking key of entity generated with format
"<partiton_prefix>.<author_first_names>-<year_of_publication>"

        key = generate_blocking_key(entity)

        # the reducer assigned is the same as the partition_prefix

        parts = key.split(".")

        reducer = int(parts[0])

        bound = reducer

        if reducer < no_of_partitions:

        # indices_None is a list of indices at which None is present
```

```
indices_None = [i for i, val in enumerate(
rep[reducer - 1]) if val is None]
# print(indices_None)
# first condition is to check if correponding list for each partition contains any None
if (len(indices_None) > 0):
rep[reducer - 1][indices_None[0]] = entity
keys_of_rep[reducer - 1][indices_None[0]] = key
else:
min_key, min_key_index = get_minimum_key(keys_of_rep[reducer - 1])
# min_entity = rep[reducer - 1][min_key_index]
if (key.upper() > min_key.upper() and entity not in rep[reducer - 1]):
rep[reducer - 1][min_key_index] = entity
 keys_of_rep[reducer - 1][min_key_index] = key
# composite key to partition by bound
new_key = str(bound) + "." + str(key)
return new_key, entity


no_of_partitions = 2
window_size = 5
rep = map_configure(no_of_partitions, window_size)
keys_of_rep = map_configure(no_of_partitions, window_size)
```

```
for value in sys.stdin:

    print('{0}\t{1}'.format(map(value)[0], map(value)[1]), end='')

print("********************************************")

for partition in rep:

    for i in partition:

    key = generate_blocking_key(i)

    bound = int(key[0]) + 1

    new_key = str(bound) + "." + key + "\0"

    print('{0}\t{1}'.format(new_key, i), end='')

# 167
```

**RepSN Reducer**

```
#!/usr/bin/env python3

import sys



def StandardSN(vals, w):

    temp = []

    temp6 = []

    arr = []

    for i in range(0, len(vals)):

    t = vals[i].split(',')
```

```
        temp6.append(t)

        temp.append(t[3].split(';'))

        for i in range(0, len(temp)-w+1):

        for j in range(i+1, i+w):

        for str1 in temp[i]:

        if str1 in temp[j]:

                l = '# '

                l += temp6[i][1]+"\t"+temp6[j][1]

                if l not in arr:

                arr.append(l)

                print(l)


no_of_partitions = 2

window_size = 5

keys_partition1 = []

keys_partition2 = []

keys_rep_sn = []

val_list_partition_1 = []

val_list_partition_2 = []

val_rep_sn = []

bound = 1


for i in sys.stdin:

        i = i.strip()
```

```
if (i[0] == '*'):
continue
key, val = i.split("\t", 1)
if (key[0] == '1'):
val_list_partition_1.append(val)
keys_partition1.append(key)
elif (key[0] == '2'):
if (key[2] == '2'):
val_list_partition_2.append(val)
keys_partition2.append(key)
elif (key[2] == '1'):
val_rep_sn.append(val)
keys_rep_sn.append(key)

w = 5  # window_size
combined_partition = val_rep_sn + val_list_partition_2
StandardSN(sorted(val_list_partition_1), w)  # comparisons
StandardSN(combined_partition[len(val_rep_sn)+1-w:], w)
first = []
last = []
curr_key = 0
```

**JobSN Phase 1 Mapper**

```
#!/usr/bin/env python3
```

```
import sys

def partition_prefix(key):

	partition_key=""

	arr1=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M']

	arr2=[ 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

	if key[0] in arr1:

	partition_key+='1'

	if key[0] in arr2:

	partition_key+='2'

	return partition_key

for i in sys.stdin:

	i=i.strip()

	entity=i.split(",")

	year=entity[5]

	author=entity[3].split(";")

	key=""

	for j in author:

	k=j.strip()

	k=k.split(" ") #first letters of author's last name

	key=key+k[0]

	mod_key=key+"-"+str(year)

	temp_key=partition_prefix(mod_key)+"."+mod_key

	value=i

	print('{0}\t{1}'.format(temp_key,value))		print('{0}\t{1}'.format(temp_key,value))
```

**JobSN Phase 1 Reducer**

```
#!/usr/bin/env python3

import sys

keys=[]

val_list=[]

bound=0

def StandardSN(vals,w):

 temp=[]

 temp6=[]

 arr=[]

 for i in range(0,len(vals)):

  t=vals[i].split(',')

  temp6.append(t)

  temp.append(t[3].split(';'))

 for i in range(0,len(temp)-w+1):

  for j in range(i+1,i+w):

   for str1 in temp[i]:

            if str1 in temp[j]:

            l='# '

            l+=temp6[i][1]+"\t"+temp6[j][1]

            if l not in arr:

            arr.append(l)

            print(l)
```

```
 c=w-1
 for i in range(len(temp)-w+1,len(temp)):
  for j in range(i+1,i+c):
        for str1 in temp[i]:
        if str1 in temp[j]:
        l='# '
        l+=temp6[i][1]+"\t"+temp6[j][1]
        if l not in arr:
        arr.append(l)
        print(l)
  c=c-1
for i in sys.stdin:
        i=i.strip()
        key,val=i.split("\t",1)
        val_list.append(val)
        keys.append(key)
w=5#window_size
StandardSN(val_list,w)#comparisons
first=[]
last=[]
curr_key=0
for j in range(0,len(val_list)):
        if j<=w-1:
        first.append(val_list[j])
```

```
            elif j>=(len(val_list)-(w-1)):
            last.append(val_list[j])
for i in range(0,len(keys)):
            k=keys[i].split(".")
            if k[0]=='1':
            curr_key=1
            elif k[0]=='2':
            curr_key=2
            if curr_key>1:
            bound=curr_key-1
            for x in first:
            s=''
            s+=str(bound)
            s=s+"."+keys[i]
            print(s+'\t'+x)#output of 1st phase and input for secind phase
            elif curr_key<2:
            bound=curr_key
            for x in last:
            s=''
            s+=str(bound)
            s=s+"."+keys[i]
            print(s+'\t'+x)#output of 1st phase and input for secind phase
```

**JobSN Phase 2 Mapper**

```
#!/usr/bin/env python3
```

```
import sys
for i in sys.stdin:
 i=i.strip()
 key,val=i.split("\t",1)
 print('{0}\t{1}'.format(key,val))
```

**JobSN Phase 2 Reducer**

```
#!/usr/bin/env python3
import sys
keys=[]
val_list=[]
bound=0
def StandardSN(vals,w):
 temp=[]
 temp6=[]
 arr=[]
 for i in range(0,len(vals)):
  t=vals[i].split(',')
  temp6.append(t)
  temp.append(t[3].split(';'))
 for i in range(0,len(temp)-w+1):
  for j in range(i+1,i+w):
   for str1 in temp[i]:
		if str1 in temp[j]:
```

```
            l='# '
            l+=temp6[i][1]+"\t"+temp6[j][1]
            if l not in arr:
            arr.append(l)
            print(l)
  c=w-1
  for i in range(len(temp)-w+1,len(temp)):
   for j in range(i+1,i+c):
            for str1 in temp[i]:
            if str1 in temp[j]:
            l='# '
            l+=temp6[i][1]+"\t"+temp6[j][1]
            if l not in arr:
            arr.append(l)
            print(l)
   c=c-1
for i in sys.stdin:
 if i[0]!='#':
            i=i.strip()
            key,val=i.split("\t",1)
            val_list.append(val)
            keys.append(key)
 elif i[0]=='#':
  print(i)
```

w=5#window_size

StandardSN(val_list,w)#comparisons

**Improved Partitioner Code**

```
import pydoop.mapreduce.api as api

from hashlib import md5

class Partitioner(api.Partitioner):

def __init__(self, context):

super(Partitioner, self).__init__(context)

self.logger = LOGGER.getChild("Partitioner")


def partition(self, key, n_reduces):

reducer_id = int(md5(key).hexdigest(), 16) % n_reduces

self.logger.debug("reducer_id: %r" % reducer_id)

if self.size>3*(500/n_reduces):#current passed size

reducer_id=partition(self,key,n_reduces)

return reducer_id
```

# SCREENSHOTS

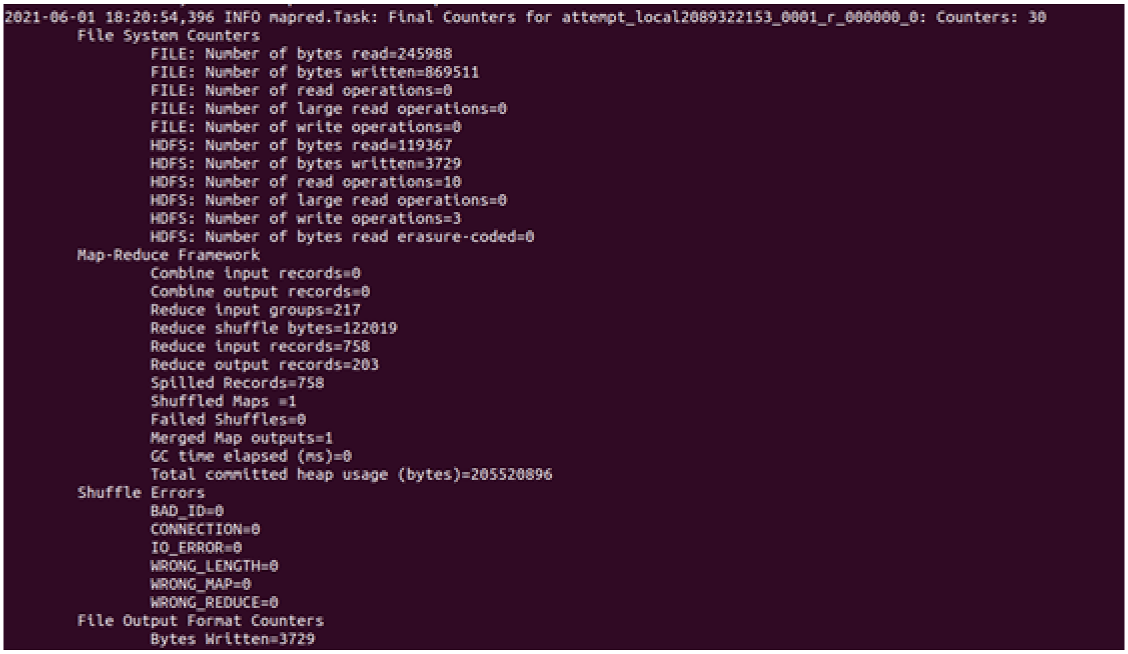
**RepSN Terminal Output**

```
Map-Reduce Framework
        Map input records=499
        Map output records=504
        Map output bytes=97817
        Map output materialized bytes=99279
        Input split bytes=86
        Combine input records=0
        Combine output records=0
        Reduce input groups=486
        Reduce shuffle bytes=99279
        Reduce input records=504
        Reduce output records=96
        Spilled Records=1008
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=9
        Total committed heap usage (bytes)=385351680
```

## JobSN Phase 1 Terminal Output

```
2021-06-01 18:20:42,706 INFO mapreduce.Job: Job job_local819508638_0001 completed successfully
2021-06-01 18:20:42,742 INFO mapreduce.Job: Counters: 36
        File System Counters
                FILE: Number of bytes read=195130
                FILE: Number of bytes written=1530210
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=159684
                HDFS: Number of bytes written=349286
                HDFS: Number of read operations=15
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=4
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Map input records=500
                Map output records=500
                Map output bytes=92577
                Map output materialized bytes=93994
                Input split bytes=85
                Combine input records=0
                Combine output records=0
                Reduce input groups=484
                Reduce shuffle bytes=93994
                Reduce input records=500
                Reduce output records=2154
                Spilled Records=1000
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=6
                Total committed heap usage (bytes)=408944640
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=79842
        File Output Format Counters
                Bytes Written=349286
2021-06-01 18:20:42,749 INFO streaming.StreamJob: Output directory: /os/jobsn/output1
```

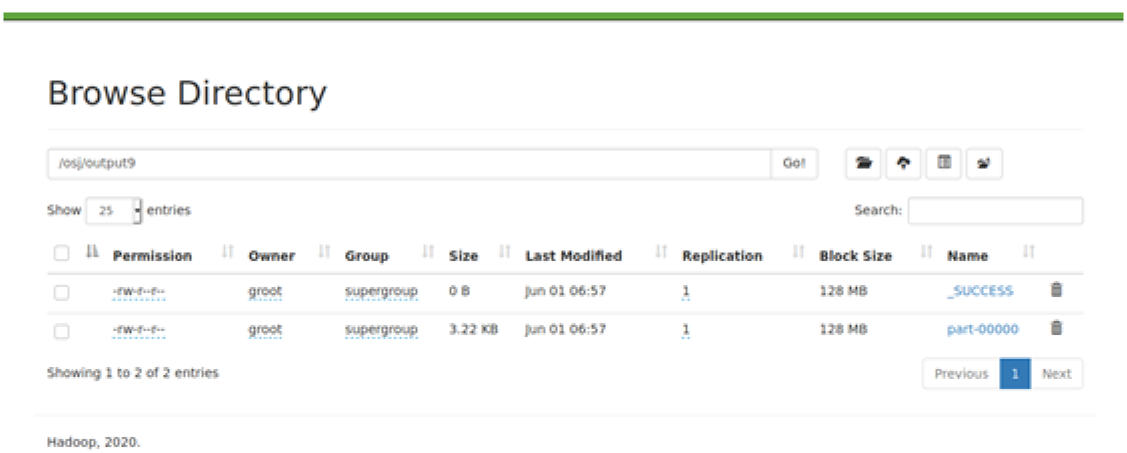
### JobSN Phase 2 Terminal Output



```
2021-06-01 18:20:54,396 INFO mapred.Task: Final Counters for attempt_local2089322153_0001_r_000000_0: Counters: 30
        File System Counters
                FILE: Number of bytes read=245988
                FILE: Number of bytes written=869511
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=119367
                HDFS: Number of bytes written=3729
                HDFS: Number of read operations=10
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=3
                HDFS: Number of bytes read erasure-coded=0
        Map-Reduce Framework
                Combine input records=0
                Combine output records=0
                Reduce input groups=217
                Reduce shuffle bytes=122019
                Reduce input records=758
                Reduce output records=203
                Spilled Records=758
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=0
                Total committed heap usage (bytes)=205520896
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Output Format Counters
                Bytes Written=3729
```

# RESULTS AND CONCLUSION

### RepSN Output from HDFS



**Sample RepSN Output**

```
# conf/vldb/FlorescuKLP97	673469
# 335465	564754
# journals/sigmod/Eisenberg96	777001
# journals/sigmod/Eisenberg96	637433
# 777001	637433
# conf/vldb/AbiteboulBBCCDMMP03	344822
# conf/vldb/AbiteboulBBCCDMMP03	journals/sigmod/BonifatiC00
# 344822	journals/sigmod/BonifatiC00
# 758376	959077
# 304230	conf/sigmod/Mohan01
# 304230	672360
# 304230	conf/vldb/Mohan02
# 304230	conf/vldb/Mohan02a
# conf/sigmod/Mohan01	672360
```

**JobSN Phase 1 Output**

```
# conf/vldb/FlorescuKLP97	673469
# conf/vldb/AbiteboulBBCCDMMP03	344822
# conf/vldb/AbiteboulBBCCDMMP03	journals/sigmod/BonifatiC00
# 344822	journals/sigmod/BonifatiC00
# 191884	615237
# 335384	conf/sigmod/PalmerF00
# conf/vldb/BelussiF95	conf/vldb/FaloutsosG96
# 950485	564770
# 950485	conf/sigmod/AbadiC02
```

```
# journals/sigmod/Snodgrass99	journals/sigmod/Snodgrass99b
# 671696	conf/vldb/ChaudhuriS01
1.1.AlanJohannesRajmohanNikiYong-2003	acm,503100,A case for dynamic view management,Yannis Kotidis; Nick Roussopoulos,ACM Transactions on Database Systems (TODS),2001
1.1.AlanJohannesRajmohanNikiYong-2003	dblp,conf/sigmod/ReveszCKLLW00,The MLPQ/GIS Constraint Database System,Yuguo Liu; Yiming Li; Pradip Kanjamala; Rui Chen; Yonghui Wang; Peter Z. Revesz,SIGMOD Conference,2000
```

**JobSN Phase 2 Output**

```
# 191884	615237

# 191949	journals/sigmod/MaximilienS02

# 219737	conf/sigmod/SistlaW95

# 223886	185828

# 227624	conf/sigmod/MumickP94

# 233352	conf/vldb/GarofalakisG02

# 233352	672356

# 253384	conf/sigmod/LiC95

# 253384	672040
```

Thus we have performed the entity matching on the DBLP/ASM dataset to match the authors using the MapReduce paradigm.

# REFERENCES

1. Load Balancing for Entity Matching over Big Data using Sorted Neighborhood

Wattamwar, Y. (2015). Load Balancing for Entity Matching over Big Data using Sorted Neighborhood.

2. Multi-pass Sorted Neighborhood Blocking with MapReduce

Kolb, L., Thor, A., & Rahm, E. (2012). Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science-Research and Development*, *27*(1), 45-63.

3. A Duplicate Detection Benchmark for XML (and Relational) Data

Weis, M., Naumann, F., & Brosy, F. (2006, June). A duplicate detection benchmark for XML (and relational) data. In Proc. of Workshop on Information Quality for Information Systems (IQIS)

4. Cost-aware load balancing for multilingual record linkage using MapReduce

Medhat, D., Yousef, A. H., & Salama, C. (2020). Cost-aware load balancing for multilingual record linkage using MapReduce. *Ain Shams Engineering Journal*, *11*(2), 419-433.