Master’s Thesis

Sampling-Based Load Balancing Strategy for Parallel Entity Resolution using Apache Spark

Author:
Nishanth Entoor Venkatarathnam
February 13, 2019

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Institute of Technical and Business Information Systems
M.Sc. Xiao Chen
Institute of Technical and Business Information Systems

External Reviewer:
Dr.-Ing. Robert Neumann
CEO, Ultra Tendency GmbH
Entoor Venkatarahnam, Nishanth:
Sampling-Based Load Balancing Strategy for Parallel Entity Resolution using Apache Spark
Abstract

Entity resolution is the process of detecting records that refer to the same entity within one dataset or across different datasets. Due to the increase in the amount of data being collected and processed by different applications in recent years, the size of datasets has increased exponentially. Processing these datasets is one of the most compute-intensive tasks which could take from several hours to days. As the size of the dataset is the biggest hurdle, existing approaches primarily focus on using blocking techniques to group similar records and distributed environment to achieve parallel entity resolution. These approaches use distributed computing frameworks like Hadoop MapReduce and Apache Spark for distributing the load across the available resources to obtain parallelism and achieve better performance. Therein, Apache Spark is proved to have a better performance because of its in-memory computation. However, when the block sizes skew, without a proper load balancing strategy, the performance of the distributed computation would be degraded. It is crucial that we maintain a balanced load across all the nodes in the distributed environment even for a highly skewed dataset. In this thesis work, we proposed a load balancing approach based on sampling for entity resolution named Sampling Based Precise Block Slicer to handle the uneven load distribution. We evaluated our approach with other state-of-the-art entity resolution load balancing approaches and it shows that our approach has a better overall performance than the other entity resolution load balancing approaches.
Acknowledgements

First, I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for providing me the opportunity to undertake thesis work in his research group.

I would like to extend my deepest gratitude to my advisor M.Sc. Xiao Chen for her valuable guidance, encouragement, feedback and for patiently answering my questions throughout my thesis.

I would like to thank Dr.-Ing. Robert Neumann for his continuous support during my Master studies and also for making time out of his busy schedule for reviewing my thesis.

Finally, I thank my family and friends for their encouragement and support to achieve my Master's degree.
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, February 13th 2019

______________________________
Nishanth Entoor
# Contents

List of Figures xii  
List of Tables xiii  

1 Introduction 1  
1.1 Motivation 1  
1.2 Goal of the Thesis 2  
1.3 Structure of the Thesis 2  

Listings 1  

2 Background 4  
2.1 Entity Resolution 4  
2.1.1 Definition of Entity Resolution 4  
2.1.2 Applications areas of Entity Resolution 4  
2.2 Entity Resolution Process 7  
2.2.1 Data Pre-Processing: 7  
2.2.2 Blocking 8  
2.2.3 Record-Pair Comparison 11  
2.2.4 Classification 13  
2.2.5 Evaluation 14  
2.3 Challenges in Entity Resolution 15  
2.4 Distributed Computation Frameworks 16  
2.4.1 Apache Hadoop MapReduce 16  
2.4.2 Apache Storm 16  
2.4.3 Apache Spark 17  
2.5 Data Skew Problem in Distributed Computing Frameworks 21  
2.6 Summary 24  

3 Methodology 25  
3.1 Global Workflow of SBPBS 25  
3.2 Sampling 27  
3.3 Precise Block Slicer 28  
3.4 Block Distribution 32  
3.5 Entity Comparison 32  
3.6 Summary 33
4 Implementation of Sampling-Based Precise Block Slicer Using Apache Spark

4.1 Block Diagram of SBPBS

4.1.1 Sampling

4.1.2 Precise Block Slicer

4.1.3 Block Distribution

4.1.4 Entity Comparison

4.2 Spark Stages and Actions of SBPBS

4.3 Summary

5 Evaluation

5.1 Experimental Setup

5.1.1 Input Datasets

5.1.2 Cluster Specifications

5.1.3 Submitting Spark Job to the Cluster

5.2 Evaluation of Sampling for Pre-Processing

5.2.1 Evaluation of Sampling Methods

5.2.2 Impact of Sampling Percentage on Processing Time

5.2.3 Comparison Between Stratified Sampling and Block distribution Matrix (BDM)

5.3 Tuning Spark Parameters

5.3.1 Optimum Partitions - Basic Approach

5.3.2 Optimum Partitions - Block Split

5.3.3 Optimum Partitions - SBPBS

5.3.4 Optimum Partitions - Block Slicer

5.4 Evaluation of Load Balancing Approaches

5.4.1 Experiment 1: Execution time

5.4.2 Experiment 2: Speedup

5.4.3 Experiment 3: Scalability

5.4.4 Experiment 4: Robustness

5.4.5 Experiment 5: Degree of Parallelism

5.5 Summary

6 Related Work

6.1 Entity Resolution

6.2 Sampling-Based Data Skew Handling Strategies

6.3 Strategies for parallel Block-Skewed Entity Resolution

6.3.1 MapReduce-Based ER Techniques

6.3.2 Spark-Based Load Balancing and ER Load Balancing Techniques

7 Conclusion and Future Work

7.1 Conclusion

7.2 Future Work

Bibliography
List of Figures

2.1 *Linus Torvalds* information from Wikipedia [1] and Britannica [2] . . . 6
2.2 Entity Resolution Process. . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Soundex character to numeric conversion table. . . . . . . . . . . . . . . 9
2.4 Fuzzy Soundex character to numeric conversion table. . . . . . . . . . . 9
2.5 Apache Spark Ecosystem [3]. . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.6 Spark's architecture [4]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.7 Execution of Spark job [5]. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.8 Hash Partitioning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.9 Range Partitioning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.10 Equal Partitioning Example. . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.1 Sampling-Based Precise Block Slicer's Global Workflow . . . . . . . . . 26
3.2 Concept of Precise Block Slicer . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1 Block Diagram of SBPBS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Example of Records with Blocking Key . . . . . . . . . . . . . . . . . . . . 36
4.3 Splitting Larger Blocks Using PBS . . . . . . . . . . . . . . . . . . . . . . . 39
4.4 Records with similarity values. . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.5 Classification step sample output. . . . . . . . . . . . . . . . . . . . . . . . 41
4.6 Actions and Stages in SBPBS's Spark Job . . . . . . . . . . . . . . . . . . . 42
5.1 Processing time for different number of partitions - Basic Approach . . 49
5.2 Processing time for different number of partitions - Block Split . . . . . 49
5.3 Processing time for different number of partitions - SBPBS . . . . . . . . 50
5.4 Processing time for different number of partitions - Block Slicer . . . . 51
5.5 Processing Time for Performance experiment . . . . . . . . . . . . . . . . 52
5.6  Processing Time for speedup experiment . . . . . . . . . . . . . . . . 53
5.7  Processing Time for Scalability experiment . . . . . . . . . . . . . . 55
5.8  Block size information . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.9  Processing Time for Robustness experiment . . . . . . . . . . . . . 57
5.10 Standard deviation for partition's size. . . . . . . . . . . . . . . . . 58
5.11 Standard deviation for partition's Processing Time. . . . . . . . . . 59
List of Tables

2.1 Example Team Members data 1. \hspace{1cm} \hspace{1cm} 5
2.2 Example Team Members data 2. \hspace{1cm} \hspace{1cm} 5

5.1 Test datasets with various sizes used for evaluation. \hspace{1cm} 44
5.2 Cluster Specification. \hspace{1cm} \hspace{1cm} 44
5.3 Predicted threshold value. \hspace{1cm} \hspace{1cm} 46
5.4 Sampling time. \hspace{1cm} \hspace{1cm} 46
5.5 Evaluation of SBPBS's processing time for different sampling percentages. \hspace{1cm} \hspace{1cm} 47
1. Introduction

In our digital world, it is quite common to represent one real-world entity in multiple ways. It is caused due to human errors while collecting the data, misinterpretation, using abbreviations and so on [6]. This decreases data quality and this will provide a wrong insight when the data is used for analysis purpose. This problem can be solved by using a process called Entity Resolution (ER).

ER which is detecting records that refers to the same entity across different datasets [7] with millions of records, is one of the most compute-intensive processes. This process involves cleaning and integrating the data to improve data quality [8]. As the amount of data that is being collected is increasing rapidly every day the ER process sometimes takes several hours to days to process huge datasets. Since a huge amount of data (big data) needs to be processed, a distributed (cluster) computing frameworks like Hadoop MapReduce and Apache Spark would be a suitable solution to achieve faster computation.

1.1 Motivation

This thesis work focuses on the challenges in ER which is a data-intensive as well as a compute-intensive process. Even though a huge amount of data is being processed with the intention to process the data faster it is also equally important to improve data quality. To understand the criticality of data quality, let us consider the application areas of ER. They are the national security analysis to counter terrorism, integrate census data, match biometric data during a crime investigation. Since it handles huge datasets and falls under the roof of big data, we need a solution to process datasets but at the same time, it should also improve data quality.

As we discussed above, the biggest challenge in the ER process is its data size. In this process of identifying duplicate records, each record has to be compared with every other record in the dataset (a cartesian product). This leads to $O(n^2)$ complexity which is highly inefficient for larger datasets. Blocking techniques (discussed in Section 2.2.2) are used in order to reduce the search space and the complexity [9]. These techniques group all the similar records in the dataset into a block and
compares each record with other records only in that block. Even after using these blocking techniques the processing time could vary from days to weeks [10]. This could be improved by using distributed computing.

In order to employ distributed computing, a cluster computing framework like MapReduce, Apache Spark or Apache Storm could be used. These frameworks result in an unbalanced distribution of load across the nodes in the cluster due to data skewness and partition strategy used in distributed computing frameworks (discussed in Section 2.5). This causes the overloaded nodes to run for a longer period of time while the other nodes are idle which leads to higher operating costs with decreased cluster efficiency.

In this thesis, we understand the data skew problem and build a load balancing algorithm for ER in a distributed computing environment in order to improve the performance and to reduce the overall processing time.

1.2 Goal of the Thesis

The primary goal is to build a better load balancing algorithm that could distribute the load almost equally across the cluster within a short period of time. To achieve this goal, we divided the work as shown below:

- Studying distributed computing framework's default load distribution strategies, finding the root cause of data skewness and exploring other reasons for increased processing time.

- Building a load balancing algorithm for the ER to overcome the data skew problem and improving the performance of the load distribution algorithm.

- Finally, evaluating the execution time, speedup, scalability, robustness, and degree of load distribution of our approach with other state-of-the-art ER load balancing approaches namely Block Slicer [11] and, Block Split [12] and basic ER approach without load balancing using synthetic datasets.

1.3 Structure of the Thesis

This thesis report is divided into 7 chapters including this chapter. The chapters are ordered as follows:

Chapter 2. In Chapter 2, we discuss the concepts of Entity Resolution, it’s application areas, a brief explanation of the entity resolution process, problems in Entity Resolution process and distributed computing frameworks. This provides the necessary knowledge to understand the whole thesis.

Chapter 3. In Chapter 3, we introduce our proposed approach named “Sampling-Based Precise Block Slicer” to solve problems encountered in Entity Resolution process on a distributed environment.

Chapter 4. In Chapter 4, we explain in detail how we implemented the aforementioned proposed approach using Apache Spark framework.
1.3. Structure of the Thesis

Chapter 5. In Chapter 5, we evaluate our approach along with other ER approaches Basic approach (Section 2.2), Block Slicer [11] and Block Split [12] with various experiments to examine the performance of our approach.

Chapter 6. In Chapter 6, we discuss the related research work. Here, we present the approaches used to solve a similar problem. This provides a latest update about the research in this area.

Chapter 7. In Chapter 7, we conclude our work by discussing the results and future work.
2. Background

In this chapter, we discuss the concept of Entity Resolution (ER) (Section 2.1), the ER process (Section 2.2), challenges faced in the ER process and methods used to overcome those problems (Section 2.3).

2.1 Entity Resolution

In this section, we discuss the definition of Entity Resolution (Section 2.1.1) and its application areas (Section 2.1.2).

2.1.1 Definition of Entity Resolution

Entity Resolution is the process of identifying duplicate records or otherwise called as identifying records referring to the same real-world objects in a single dataset or across different datasets [7]. It is also called as record linkage, de-duplication, entity matching, object identification [13], data matching, identity uncertainty [14], identity resolution etc. It eventually improves the data quality [8].

In order to understand ER in detail, let’s take a look at Table 2.1 and Table 2.2. From these tables, we can see that both the tables have the same columns \textit{RecordID}, \textit{Name}, \textit{City} and, \textit{Age}. Also, we find that there are a couple of errors in the \textit{Name} column. \textit{Rec1} in Table 2.1 and Table 2.2 refers to the same person but the name is misspelled in one of the records. Likewise, \textit{Rec2} in Table 2.1 and \textit{Rec3} in Table 2.2 refers to the same person. When these two tables are merged together based on the names, then there will be multiple duplicate records for the same person. Hence these duplicate records should be removed or merged. Since it is a small example we are able to easily find the duplicate records but in the real world the size of data will be huge and it is not feasible to do it manually. So we employ the ER process to solve this problem.

2.1.2 Applications areas of Entity Resolution

To understand the need for ER, we discuss the most common applications where Entity Resolution is used. For example, data integration, counter-terrorism measures, integrating census data, fraud detection, web search and so on.
### 2.1. Entity Resolution

<table>
<thead>
<tr>
<th>RecordID</th>
<th>Name</th>
<th>City</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec1</td>
<td>Prasanth</td>
<td>Chennai</td>
<td>26</td>
</tr>
<tr>
<td>Rec2</td>
<td>Victor</td>
<td>Magdeburg</td>
<td>27</td>
</tr>
<tr>
<td>Rec3</td>
<td>Frank</td>
<td>Berlin</td>
<td>36</td>
</tr>
<tr>
<td>Rec4</td>
<td>Jamie</td>
<td>Cardiff</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 2.1: Example Team Members data 1.

<table>
<thead>
<tr>
<th>RecordID</th>
<th>Name</th>
<th>City</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec1</td>
<td>Prashanth</td>
<td>Chennai</td>
<td>26</td>
</tr>
<tr>
<td>Rec2</td>
<td>Frank</td>
<td>Magdeburg</td>
<td>27</td>
</tr>
<tr>
<td>Rec3</td>
<td>Viktor</td>
<td>Berlin</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 2.2: Example Team Members data 2.

- **Data Integration:** Data Integration is the process of integrating or combining two or more data sources into a single data source [15]. The major problem in data integration is entities from different data sources refer to the same real-world entity. This results in record duplication, in order to remove these duplicate records all the similar or duplicate records are merged together into a single entity. The ER process is used to find the duplicate records before they are merged.

Figure 2.1 shows data collected from two different sources Wikipedia\(^1\) and Encyclopædia Britannica\(^2\), they both could represent the same real-world entity in different ways. It gives the information about the creator of Linux Linus Torvalds from both the sources. Encyclopædia Britannica has only limited information about him, where as Wikipedia contains additional information such as residence, known for, spouse(s), parent(s) and relatives. Hence, these two entities from two different data sources referring to the same real-world entity can be merged into a single entity to get a unified view.

- **Counter-Terrorism Measures:** The role of National Security Agencies is to protect people from cyber-criminals and terrorist attacks [16]. Today, the major security threats to most of the countries are due to the terrorist organizations that operate in a decentralized way.

One of the widely used techniques to understand the individuals and the structure of these organizations is social network analysis. Entity Resolution can be used to detect the unexpected connections between different entities which might be referring to the same real-world entity. In some cases, it can also help to connect the dots between two different real-world entities with some common attributes they share. For example same the address, same employer etc. Thus, ER plays a vital role to counter terrorism.

\(^1\)https://www.wikipedia.org/
\(^2\)https://www.britannica.com/
Figure 2.1: *Linus Torvalds* information from Wikipedia [1] and Britannica [2].
• **Integrating Census Data:** Census data which includes an individual’s personal information such as given-name, surname, address, income etc., are updated periodically by the organizations responsible for collecting census data. The organizations those collect the census data might change from time to time or there could be some errors while collecting census data. This could be a problem when the latest census data needs to be merged with the previously existing data. Also sometimes the individual could have moved to a new address, so it is important to merge the census data and make sure there aren’t any duplicate records that refer to the same individual person. ER techniques are widely used to find out the duplicate records from different census datasets and later they are merged.

• **Others:** There are also several other applications apart from the those mentioned above. They are such as comparing online products, web search, genome databases, fraud detection, bibliographic citation [15]. Apart from ER being used in several applications it also helps to clean the dataset and to improve the quality of the data.

### 2.2 Entity Resolution Process

ER process is primarily divided into five steps. They are data pre-processing, blocking, record-pair comparison, classification, and evaluation as shown in Figure 2.2. In this section, we will discuss all these five steps in detail. This standard approach of not using any load balancing strategy in Entity Resolution process can be called as Basic approach.

![Figure 2.2: Entity Resolution Process.](image)

#### 2.2.1 Data Pre-Processing:

The datasets (data sources) in the real world are incomplete, contains null values, missing some attributes, missing important attributes, attributes with errors and so on. Before performing the actual processing on the dataset we need to clean the dataset by removing the records that contain only null values. This improves the quality of the data and also helps to obtain better results. This process of cleaning the data is called as data preparation [17] or data pre-processing.
2. Background

2.2.2 Blocking

This step is the start of the ER process, once the dataset is pre-processed we get a clean dataset to proceed further with our ER process.

The costliest operation in ER is the comparison part where each record should be compared with every other record to find out the duplicate records. If blocking is not done, then the comparisons performed would be \( O(n^2) \) comparisons for \( n \) records. For instance, if we have 50 records in a dataset on which we need to perform ER, then the number of comparisons without blocking will be \( 50 \times 50 \) 2500 comparisons. Whereas if all the similar records are blocked then we need to compare only the records inside those blocks, so it substantially reduces the search space which eventually results in a lesser number of comparison.

Blocking-Key Values (BKV) Generation Methods

In this section, we discuss various methods used in ER for generating blocking-keys. As we discussed, blocking is the most important part in the ER process. But the most important part of blocking is generating the BKV. This determines the quality of comparison record-pairs. This generated BKV is added as an attribute to each record in the dataset.

BKV is generated by choosing one or more attributes from the dataset and applying phonetic encoding function on those attributes. The attributes chosen for BKV generation is done by a domain expert. These phonetic encoding functions convert the string attributes to numeric values to compensate the errors. We will discuss three phonetic encoding techniques Soundex, Metaphone and Double Metaphone below:

1. **Soundex**: Soundex is one of the most widely used phonetic encoding techniques because of its simplicity and computational efficiency [18, 19]. This technique can only be used with American English language. It converts the string into Soundex code, below are the rules followed:
   - The first letter of the string is not converted into an equivalent Soundex code. It is taken as it is and it will be the first letter (prefix) of the code.
   - The occurrence of a, e, i, o, u, h, w and y which are code 0 (from Table) are removed.
   - Convert the remaining characters other than those from code 0 based on Table shown in Figure 2.3.
   - Once the whole string is converted into a code, remove the repetitions of a number (keep only the first occurrence).
   - Limit the size of the Soundex code to four characters with one alphabet (prefix) and three numbers. If the resulting code is less than the four characters add zeros at the end, else if the resulting code is more than four characters remove the extra characters(numbers). - For example: Soundex code of ‘Nishanth’ will be ‘N253’.

2. **Metaphone**: Metaphone algorithm is a better phonetic algorithm than Soundex for the English language [17]. This algorithm uses pronunciation information to
2.2. Entity Resolution Process

- Keep the first letter and remove vowels after the first letter.
- Replace alphabets with 16 constants except for the first letter.
- For example: Metaphone of ‘Nishanth’ will be ‘NXNO’.

3. Double Metaphone: The Double Metaphone is the next version of the Metaphone algorithm [20]. As we have already discussed Metaphone algorithm only supports English names, whereas the Double Metaphone algorithm also provides support for Asian and European names [21]. Unlike the Metaphone algorithm which generates only one phonetic name, Double Metaphone algorithm generates two phonetic names. For example, Double Metaphone of ‘Nishanth’ will be ‘NXN0’ and ‘NXNT’.

4. Fuzzy Soundex: Fuzzy Soundex is similar to Soundex algorithm and it uses q-gram for pre-processing [18]. It generates q number of sub-strings when q is two it is called as bigrams. When q is three it is called as trigrams. This technique replaces the q-gram (sub-string) characters with numeric values based on the table shown in Figure 2.4. This approach using Q-gram gives better results than the Soundex algorithm [22].

Blocking Techniques

In Section 2.2.2, we discussed how to generate the blocking-key for all the records in the dataset. Now, we discuss the blocking techniques used to create blocks by
grouping records. In order to perform blocking on a set of records, we need to use different blocking techniques based on the type and the structure of the data. These techniques generate a blocking-key for every record and they are grouped into blocks. Then, only the records within a block are compared with each other i.e., records with same the blocking-key. Some of the blocking techniques are:

1. **Standard Blocking Algorithm**: This blocking technique is widely used in the field of Data Matching\cite{7}. This technique generates a blocking-key for each record and the records with the same blocking-key are inserted into the same block. An inverted index structure \cite{23,24} with blocking-key as the key is created in order to improve efficiency. The lesser the difference between the mean of the block sizes and the size of the largest block, the better the records are uniformly distributed. Let us assume two datasets $d_1$ and $d_2$, total number of blocking-key common as $t$. Thus, each block will contain $d_1/t$ and $d_2/t$ records respectively. Then, the total number of record-pairs generated for both the datasets $d_1$ and $d_2$ is taken as $p$, then $p$ is

$$p = t \left( \frac{d_1}{t} \cdot \frac{d_2}{t} \right) = \frac{d_1d_2}{t}$$ \hspace{1cm} (2.1)

One dataset with $d$ records, $t$ blocking-keys, $p$, is

$$p = t \left( \frac{d}{t} \cdot \frac{d-1}{t} \right) / 2 = \frac{d(d-1)}{2t} \hspace{1cm} (2.2)$$

2. **Sorted Neighbourhood Blocking**: Sorted Neighbourhood blocking is one of the alternative approaches for Standard Blocking Algorithm \cite{25,26}, similar to Standard Blocking Approach’s blocking-key Sorted Neighbourhood generates sorting-key. Once the sorting-key is generated the records in the dataset are sorted based on the generated key. In this approach, it is assumed that once the records are sorted all the similar records will be closer to each other. In case if there are multiple datasets (data sources), the datasets are merged after generating sorting-key and then the combined dataset is sorted.

As we have seen in Standard Blocking algorithm only a subset of records (block) is taken to generate comparison record-pairs, whereas in this blocking technique a sliding window is used. This sliding window is moved through the records in the datasets. Comparison record-pairs are generated from the records that fall in the sliding window. The fixed window size is a huge limitation for this approach. If the window size is too small, then some true record-pairs could be ignored.

3. **Q-gram Based Blocking**: Both Standard Blocking Algorithm and Sorted Neighbourhood Blocking performs better when there is less disturbance in the dataset. But when the disturbance and errors in the dataset are high those blocking techniques are not efficient and result in a poor record-pair generation.

Q-gram Based Blocking technique also generates a blocking-key in a similar way as Standard Blocking. But instead of using the whole blocking-key to
create a block, this approach generates index keys for each variant of the blocking-key. This results in placing one record in more than one block [9]. A list of \textit{q-grams} (blocking keys) are generated from the original blocking key. Each q-gram will be a sub-string of the original blocking-key and the size of the q-grams can be specified by the user. It is also called as \textit{n-gram} [27], when the size of q-grams i.e., \( q \) is 2 then it is called as \textit{bigrams} or \textit{diagrams} [28]. Whereas when \( q \) is 3, then it is called as \textit{trigrams} [29]. For example, if there is a blocking-key as ‘germany’ and we choose the size of the q-gram as 2 (bigrams). Then the list of q-grams (bigrams) will be [‘ge’, ‘er’, ‘rm’, ‘ma’, ‘an’, ‘ny’].

The comparison record-pairs generated using this blocking technique are more accurate, the downside of this approach is its time consumption. Since one record is placed in multiple blocks, the number of comparison record-pairs generated is higher than the record-pairs generated in the above two approaches and thus increasing the comparison (computation) time.

4. \textit{Learning Optimal Blocking Keys:} Usually, in the blocking techniques discussed above the blocking keys are determined by specific domain experts. Determining the optimal blocking keys will increase the quality of the comparison record-pairs and reduce the search space, but it is not an easy task especially for larger datasets [27, 30].

An automated approach can be used to define the blocking keys, machine learning algorithms [31, 32] based on supervised learning are used to automate the process of blocking key definition. These algorithms use true match and non-true match pairs as the training data to generate automatic blocking keys.

5. \textit{Others:} The above-discussed blocking techniques are the most widely used techniques in the ER process. Suffix-Array Based Blocking, Map-Based Blocking etc., are few other blocking techniques used in the ER process.

2.2.3 Record-Pair Comparison

This step is the heart of Entity Resolution process. Once the blocks are created based on the blocking-key as discussed in Section 2.2.2, we generate comparison record-pairs for all the records within that block. Then, these record-pairs should be compared to find out whether the record-pair is a \textit{match} or \textit{non-match}. This record-pair comparison is the most compute-intensive task in the whole ER process.

The comparison of both the records in the record-pair is done by comparing each attribute or field from both the records to calculate the similarity between them. There are different techniques for calculating similarity metrics based on sound, character, etc. Phonetic encoding techniques discussed in previous section falls under sound-based similarity metrics. Since we need to find the similarity between records with string fields, we need a technique to calculate the similarity between strings. In order to do so, we can use character-based similarity measures [17]. Some of the character-based similarity measures are discussed below:
1. *Exact Similarity Measure:* This measure is either a *match* or *non-match* metric. It will be a match only if all the characters in the string match. Then, the output will be 1. Else the output will be 0 which is a *non-match* [7].

2. *Levenshtein Distance Similarity Measure:* Levenshtein Distance is one of the edit-based techniques for calculating distance between two strings [33]. The Levenshtein Distance is the number of edit operations required to transform a string to other string that it is being compared with. For example, if string A is compared with string B. Then the number of edit operations required to transform string A to string B is the Levenshtein Distance. The edit operation rules include:
   - Insert a character in the string.
   - Delete a character in the string.
   - Replace a character with another character in the string.

Using dynamic programming algorithm [34], the edit distance between both the strings is calculated by taking the edit operations count.

For example, if we calculate the Levenshtein Distance between *'tablet'* and *'tables'* it will be 1.

3. *Hamming Distance Similarity Measure:* Hamming Distance is a edit-based similarity technique similar to Levenshtein Distance technique. Unlike Levenshtein Distance, Hamming Distance transform one string into another only by replacing a character with another character in the string which is only one of the edit operations allowed in Levenshtein Distance technique. This methods is used only where the two strings are of same length. The distance ‘d’ for string X and string Y ranges in between 0 ≤ d(X,Y) ≤ |X| [33]. For example, Hamming Distance between *‘Nishanth’* and *‘Prasanth’* is 4.

4. *Jaro-Winkler Distance Similarity Measure:*

   **Jaro Distance Similarity Measure:** Matthew Jaro developed the algorithm to calculate the similarity between the names (string) and this is also an edit-based similarity measure. In order to understand how to calculate Jaro distance, let’s assume two strings namely x and y. Let the lengths of x and y be L_x and L_y respectively. This algorithm calculates the common characters from both the strings x and x within a given window positions, let’s take common characters as c. Finally let’s consider half the number of transformations required to convert x to y as t. Then the jaro distance is calculated as [35]:

   \[ Jaro(x, y) = \frac{1}{3} \left( \frac{c}{L_x} + \frac{c}{L_y} + \frac{c - t}{c} \right) \]  

   **Jaro-Winkler Distance Similarity Measure** is based on Jaro Distance algorithm but a few changes are made to create Jaro-Winkler algorithm. They are:
   - Prefix of the strings are given a high preference. If both the strings have the same prefix, then the similarity between them will be increased.
   - The similarity between two strings is also increased by applying Winkler algorithm to Jaro Distance algorithm as shown in Equation 2.4 where p is the length of prefix which ranges from 1 to 4.
2.2. Entity Resolution Process

\[
sim_{\text{Winkler}}(A, B) = Jaro(x, y) + \frac{p}{10} (1.0 - Jaro(x, y)) \quad (2.4)
\]

2.2.4 Classification

The similarity measure calculated for the record-pairs in the record-pair comparison step is used to classify the record-pairs as ‘match’ or ‘non-match’. This type of classification is called as binary or binomial classification. Let’s discuss about three classification techniques namely threshold-based classification, rule-based classification and supervised classification.

1. Threshold-based Classification This technique is used to classify whether a record-pair is a ‘match’ or ‘non-match’ by summing up the similarity measure values for all the fields obtained in record-pair comparison step. The total sum of similarity measure values is called as \(\text{Sum}_{\text{Sim}}\) [7]. \(\text{Sum}_{\text{Sim}}\) is compared against the threshold value \(t\), if \(\text{Sum}_{\text{Sim}}\) is greater than or equal to \(t\) then it will be labelled as a ‘match’. Else if \(\text{Sum}_{\text{Sim}}\) is less than \(t\) then it is labelled as ‘non-match’. Apart from labels ‘match’ or ‘non-match’ threshold-based classification also can have a third label as ‘potential-match’. For a record pair \((x, y)\), the classification can be expressed as:

\[
\text{Sum}_{\text{Sim}}(x, y) \geq t \Rightarrow (x, y) \Rightarrow \text{Match}
\]

\[
\text{Sum}_{\text{Sim}}(x, y) < t \Rightarrow (x, y) \Rightarrow \text{Non-Match}
\]

If ‘potential-match’ label needs to be added, then we should have a lower limit \(t_{\text{low}}\) and an upper limit \(t_{\text{up}}\) for the threshold value \(t\). In this case the classification can be expressed as:

\[
t_{\text{low}} < \text{Sum}_{\text{Sim}}(x, y) < t_{\text{up}} \Rightarrow (x, y) \Rightarrow \text{Non-Match}
\]

In this technique, each field is given the same weight. This makes both the most and least importance fields to be considered as same. In order to overcome this, a weighted threshold approach can be employed by assigning weight to each field based on their importance [7].

2. Rule-based Classification Rule-based classification determines a record-pair as a ‘match’, ‘non-match’ or ‘potential-match’ based a certain rules [25]. These rules are applied on the similarity measure values calculated in record-pair comparison step (Section 2.2.3). These rules are created by combining conjunctions, disjunctions, and negations.

3. Learning-based Classification In Learning-based Classification, different machine learning algorithms are used to develop a probabilistic classifier. It is then trained with a training dataset containing true match and non-match pairs and then the classifier is used for classifying the match pairs. This increases classification accuracy but it will be a time consuming process because the time spent on training the classifier. But this can be used where the classification quality is more important than the processing time.
2.2.5 Evaluation

Evaluation is the final step in the entity resolution process. In this step, we evaluate the data after classification. Ground-truth data which is also called ‘gold-standard’ data [7] of these records is required to evaluate the quality of the data. Gold standard data has the information regarding the correct match and non-match of the records. This data helps us to evaluate the accuracy of the records classified in Section 2.2.4.

We can assign the classified records into four categories which is called as confusion matrix. The four categories are as follows [36]:

- **True positives**: The record-pairs that fall into this category are the exact match pairs, they are classified as a match in the classification step and they are also true match record-pairs. These record-pairs refers the same real-world entities.

- **False positives**: The record-pairs in false positives categories are those record-pairs classified as a match in the classification step but are not actual match pairs when they are compared with gold-standard data.

- **True negatives**: The record-pairs in this category are classified as a non-match in the classification step and they are also actual non-match. These record-pairs does not refer to the same real-world entity.

- **False negatives**: The record-pairs in false negatives are classified as non-match in the classification step but they are real match record-pairs. It means that the record-pairs refers to the same real-world entity but it is classified wrong in the classification step.

The quality measures such as precision, recall and F-measure are calculated to evaluate the quality of the record-pairs classified based on true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

- **Precision**: It is the most widely used quality measure to estimate the quality of the match pairs [23]. Precision is measured as the ratio of true positive (TP) matches to the sum of true positive (TP) and false positive (FP) matches i.e., TP+FP. In other words it can be called as the ratio of true matches to total matches classified in the classification step.

\[
\text{precision} = \frac{TP}{TP + FP} \tag{2.5}
\]

- **Recall**: It is one of the most widely used quality measure after precision for evaluating data in data retrieval techniques [23]. Recall is otherwise called as true positive rate or hit rate. Recall is calculated by taking the fraction of true positive (TP) to the sum true positive (TP) and false negative (FN) i.e., TP+FN which could be considered as true matches obtained after comparing with gold-standard data. It is used to calculate the proportion of actual true matches from the total matches those are classified in the classification step.
2.3 Challenges in Entity Resolution

In this section, we discuss about the challenges faced in ER process. Time consumption is the primary challenge in ER process. As a result of this it is difficult to handle larger datasets and it also suffers due to data heterogeneity. A brief explanation about these two problems, their consequences and solutions to overcome these problems are given below:

- **Size of the Dataset:** As we have already discussed in the previous sections, huge datasets are used in ER process. It could sometimes take several hours to several days to process the whole dataset depending up on the size and complexity of the data. The huge processing time can be reduced by using parallelization. This parallelization can be achieved either by using a parallel database or by a distributed computing framework. We discuss these two approaches below:

  1. **Parallel Databases:** Parallel Databases achieve better performance by parallelizing the database operations such as building indexes, loading data and evaluating queries [38]. This parallelism is achieved by using multiple processors (CPUs) and disks in parallel. Parallel Databases can be implemented using three different types of architectures. First, the processors share the main memory but with their own disk. Second, processors share the disk but with their own main memory. Third, they share nothing. This proves to give better performance than a centralized database.

  2. **Distributed Computing Frameworks:** While using distributed computing frameworks, the task (data to be processed) is distributed across a cluster of computers. These computers coordinate with each other to complete the task faster. In the last decade, distributed computing frameworks has drawn a lot of attraction for its lightning fast performance. Most of the distributed computing frameworks use in-memory computation in order to achieve better performance.

From the above two approaches, we understand that distributed computing will be an ideal choice for ER to achieve better performance times because of its processing power. Since parallel databases takes care of only the operations related to databases, it will not provide much improvement in performance for ER because ER is a compute-intensive process which requires a faster computing solution. We discuss about few distributed computing frameworks in Section 2.4.

\[
\text{recall} = \frac{TP}{TP + FN} \quad (2.6)
\]

- **F-measure:** It is defined as the harmonic mean of both recall and precision [37]. It is also called as f-score. Equation 2.7 shows how f-measure is calculated. It shows that f-measure is directly proportional to precision and recall.

\[
f\text{-measure} = \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (2.7)
\]
• *Data Heterogeneity:* Heterogeneous data is quite common in a real-world dataset. Even though it is highly difficult to completely remove the heterogeneity. We can improve quality of the dataset by properly pre-processing the dataset.

## 2.4 Distributed Computation Frameworks

In this section, we discuss about few commonly used distributed computing frameworks: Apache Hadoop MapReduce (Section 2.4.1), Apache Storm (Section 2.4.2) and Apache Spark (Section 2.4.3).

### 2.4.1 Apache Hadoop MapReduce

Apache Hadoop MapReduce is a distributing computing framework for batch processing of big data on a cluster. This framework allows you to parallelize the data that needs to be processed across a multi-node cluster. It can be used to process both structured and unstructured data that is stored in Hadoop Distributed File System (HDFS).

MapReduce consists of three steps. They are:

- **Map:** The nodes responsible for processing are called as *worker* nodes. Where as task assignment is done by *master* node. These worker nodes apply the map function which is a transformation applied to the input data. The intermediate result is stored into the temporary memory. The master makes sure that only one copy of the input data is processed.

- **Shuffle:** The worker nodes redistributes the *intermediate results* from the map step based of their keys. This allows all the records with the same key to be processed on the same node.

- **Reduce:** The worker nodes process the shuffled data from the shuffle step in parallel.

The MapReduce jobs on the cluster are coordinated by Hadoop Yet Another Resource Negotiator (YARN). The primary tasks of YARN are resource management on the cluster and job scheduling. Hadoop MapReduce programs can be written in Java, Python and Ruby.

### 2.4.2 Apache Storm

Apache Storm is a distributed computation framework mostly written in Clojure programming language for processing data streams in real-time. Storm is similar to Hadoop MapReduce, this does the same what MapReduce does for batch processing but here it is done for stream processing. Even though it is primarily developed for stream processing it also supports batch processing. Storm can be integrated with Hadoop in order to achieve better throughputs.

Like Hadoop, Storm also has Master node and Worker node. Master node is responsible for distributing and assigning tasks across the worker nodes. Whereas worker
nodes are the nodes where the actual data processing takes place. The coordination between the master node and worker node is taken care by Apache Zookeeper.

Apache Storm comprises of four components. They are:

- **Topology**: It can be defined as a network of *spouts* and *bolts* which are similar to map and reduce tasks in MapReduce. Unlike MapReduce job the topology doesn’t stop once the job is finished, it has to be killed manually.

- **Stream**: It is the basic abstraction in Storm and it is an unbounded sequential set of tuples. These storm tuples can have one or more fields with a schema. An ID is assigned to each stream when they are declared.

- **Spout**: Spout is the point of entry for data into the topology. It is responsible for continuously receiving the data from the source. It then transforms the data into sequence or stream of tuples and send them to bolts for further processing.

- **Bolt**: The logic for processing the streams from spouts resides here. Bolt is where the actual processing is done in the topology, they either process the streams or emit the streams to other bolts for processing. They also send the data to storage volumes for persisting. It is also possible to run functions, apply filters on the tuples and to join streams.

Storm is easy to implement and it is possible to implement it with any programming language.

### 2.4.3 Apache Spark

Apache Spark is an open-source, in-memory, fault-tolerant cluster computing framework for processing big data in a parallel fashion to provide a lightning fast data processing. It is a fast and general-purpose cluster computing framework [39] and it can be used for both batch and stream processing. It helps to distribute the workload across all the nodes in the cluster to achieve parallelism and to reduce the overall processing time considerably. This will be more suitable for processing data that would take several hours or days if it is processed on a single node. It not only reduces the processing time just because of parallelism but also makes the computation faster than usual due to it’s in-memory computation where it stores the temporary results in-memory instead of writing them to disk. It employs lazy evaluation which is also one of the factors for achieving faster computation. The primary features of Apache Spark are speed, ease of use, generality and runs everywhere [3]. Let’s discuss these features in detail.

**Speed**: Apache Spark processes the workloads 100x faster than Apache Hadoop with the help of a query optimizer, state-of-the-art Direct Acyclic Graph(DAG) scheduler and a physical execution engine [3].

**Ease of use**: Spark applications can be written in many different languages that includes Java, Scala, Python, R and SQL. Spark extending it’s support for R and SQL attracted more data analysts and data scientists to use Spark. Except Java
for the remaining languages spark can be used interactively from their respective shells. It provides more than 80 high-level operators to easily build parallel processing spark applications. It also provides rich API’s namely RDDs, Dataframes and Datasets.

**Generality:** Spark provides a wide range of libraries that includes MLlib for machine learning, SQL and Dataframes, Spark Streaming and GraphX. This opens up the space for developing spark applications by simply extending these libraries and combining them seamlessly in the Spark application.

**Runs everywhere:** Spark runs on multiple platforms that include Kubernetes, Hadoop, Apache Mesos or cloud platforms such as Microsoft Azure, Amazon Web Services etc. It is also possible to run it in a standalone mode on a local computer or on a cloud instance. It can access data from a wide range of data sources such as Apache HBase, HDFS, Apache Cassandra, Apache Hive, Alluxio and so many other data sources.

**Evolution of Apache Spark**

Spark is one of the many top projects developed at UC Berkeley. In 2009, Matei Zaharia who was specializing in the area of distributed systems, big data and cloud computing started the Spark project at UC Berkeley’s AMPLab during his PhD studies. He then open sourced it in the next year.

Spark was initially developed as an alternative for Hadoop as it’s performance was slow for Machine Learning because of its data exchange with HDFS between each iteration. It became of massive success because of its seamless integration with Hadoop ecosystem. It also got its reach because of its lightning fast in-memory computation. Once the project started to grow at a rapid pace it was donated to Apache Software Foundation in 2013 and it became a top-level Apache Project in February 2014. Three months later in May 2014 Apache Spark API was released.

**Apache Spark APIs**

Apache Spark provides three APIs or data abstraction interfaces. They are Resilient Distributed Dataset (RDD), DataFrame and Dataset. RDD is available since Spark 1.0, DataFrame is introduced by Spark in Spark 1.3 and Dataset is available from Spark 1.6 release. We will discuss about these three APIs briefly.

**RDD:** RDD is the basic abstraction in Spark, it is an immutable collection of data elements distributed and partitioned across multiple nodes in a cluster. Even the other two abstraction interfaces DataFrames and Datasets will be converted into RDDs before they are processed. The data objects distributed across the cluster are processed parallely and it is managed by a lineage graph called as Direct Acyclic
Graph (DAG). DAG keeps track of these partitions and the transformations performed on these partitions. It helps to reconstruct the partitions if they are lost. RDDs are preferred when we need have more control over the data we process.

**DataFrame:** Similar to RDD, DataFrame is a distributed immutable collection of data objects but it provides a schema by considering the data as a table with column names and data type information. This allows us to run SQL queries on DataFrames. The main advantage of DataFrame is its huge performance boost compared to RDDs. This performance boost is achieved by two of its powerful features: Custom Memory management and Catalyst Optimizer. But the DataFrame comes with a downside, which is its lack of type safety that could cause failures during runtime.

**DataSets:** Dataset API is created by extending DataFrame API to provide the best features of both DataFrame and RDD APIs. This abstraction comes with compile time type safety and with the powerful features of DataFrame that boost the performance.

**Apache Spark Ecosystem**

The Apache Spark ecosystem is comprised of five components with Spark Core being the heart of the ecosystem. The other components Spark SQL, Spark Streaming, MLlib and GraphX are built on top of Spark Core. These components acts as add-ons to Spark Core and it is possible to integrate these components in a single Spark application. Figure 2.5 shows the components of Spark Ecosystem. We discuss about these components in detail below:

![Apache Spark Ecosystem](image)

**Spark Core:** As Core being the heart of the Spark, it contains the basic functionality. Core implements and manages functionalities, which include fault tolerant, scheduling, memory management, interaction with persistence layer and more. It contains Spark’s main programming abstraction i.e. resilient distributed datasets (RDD). RDDs support in-memory computations, distributed data partitions and fault-tolerant.

**Spark SQL:** The Spark SQL component is built to handle structured data processing. It allows us to perform SQL operations on structured data or to specify
the schema explicitly on the (semi-structured) data if possible and then run SQL operations on the data. But if the data is already in Hive table it will provide a full compatibility and allows to run both SQL and HQL (Hive Query Language) queries on the data. Once the data is loaded it treats the data as a table with columns and rows. It works with a variety of data sources that include Parquet, Avro, JSON, Hive, JDBC and JSON. This component provides the power of Spark Core and also the possibility to run analytical queries on the data in a distributed way.

**Spark Streaming:** Spark Streaming is a streaming engine built on top of Spark Core to provide a scalable and fault-tolerant stream processing framework. It is a potential alternative to Apache Storm because of its stability and the possibility to integrate it with other components of Spark such as MLib for Machine Learning, Spark SQL and GraphX. It creates micro-batches from the incoming streams of data and then give it to the processing system to provide a near real-time processing experience. For example, such kind of processing is employed in processing GPS data from vehicles, performing sentiment analysis on twitter tweets and so on. It can be integrated with data ingestion frameworks or tools like Apache Kafka, Apache Flume, and Kinesis.

**MLib (Machine Learning library):** MLib is developed in order to provide a scalable, fast and easy solution for Machine Learning. It comes with a suite of commonly used machine learning algorithms that includes classification, regression, collaborative filtering and clustering. From Spark 2.0 RDD API in MLib module entered the maintenance mode and Dataframe API has become the primary API for Machine Learning in Spark. So there will not be any new features added to RDD API. This decision is made because the Dataframe API is more user-friendly than RDD API.

**GraphX:** GraphX is completely a separate project developed at UC Berkeley and it was donated to Apache Software Foundation. Then it was integrated into Spark ecosystem for graphs and graph parallel-computation [40]. It is possible to perform classification, clustering, searching, pathfinding and traversal in graphs. It provides a new abstraction for Graph by extending the Spark RDD and a feature to build distributed directed multigraph with properties attached to each vertex and edge [41].

**Spark Architecture**

Spark employs a master-worker architecture with a cluster manager to coordinate between them. A physical machine is called as a node, as shown in Figure 2.6 the master node is called as driver or driver node and each worker node has one or more executors. The driver and executors run on their own Java Virtual Machines (JVM). The driver can either physically or virtually separated. For instance you can run both the driver and the executor on a single machine (horizontal cluster) or on different machines (vertical cluster) or in a mixed configuration.

The driver is the master node of a spark application and it hosts the SparkContext1. It is responsible for splitting and scheduling the tasks to run the executors. It also maintains all the required information until the spark job is completed. Executor is where the actual processing is done and it is distributed across the cluster. Apart
from processing the task assigned by driver they will also report the status of the job to the driver. Whereas the Cluster Manager takes care of the resource allocation. The cluster managers available to manage a spark cluster are Yet Another Resource Negotiator (YARN), Apache Mesos or Spark standalone mode.

### Execution of Spark Application

Once the driver starts, spark job loads the dataset into RDD, it splits the datasets into multiple partitions and stores them on the executors to process them in parallel. The number of partitions, number of executors, executor memory and so on can be given as static values or allowed to set dynamically by Spark, it is explained in detail in Section 5.1.1.

The spark program consists of actions and transformations. Transformation applies an user specified function and returns a RDD, whereas action performs all the previously defined transformations and returns something other than RDD. This process is called as lazy-evaluation.

Since the driver will not start the evaluation unless there is an action, the spark scheduler creates an execution plan for each action and then starts the spark job. Each jobs is defined as a set of stages which in turn contains a set of tasks. These tasks are executed parallely on the executors based on the execution plans created. Figure 2.7 shows how a spark job is divided into jobs, stages and tasks.

### 2.5 Data Skew Problem in Distributed Computing Frameworks

In this section, we discuss about the data skew problem that occur in the distributed computing frameworks explained in Section 2.4.

Even though the distributed computing frameworks distribute the load across the cluster, sometimes it does not significantly reduce the processing time because of
the unbalanced load among different nodes and the skewed data that needs to be processed. The cause of unbalanced load is because of their partitioning strategies: hash partitioning and range partitioning. Figure 2.8 and Figure 2.9 shows examples for range partitioning and hash Partitioning respectively where range partitioning distributes the records/tasks equally across the available partitions based on the defined range and hash partitioning distributes the load based on a hash function. For instance, consider the keys in the below figure as the keys of each task which consists of several records. Here, the hash function is considered as the length of the key for an easy understanding. Whereas in Figure 2.9, we have 9 records, 3 partitions, keys of those records range from 1 to 15, we set the range partitioning from 1 to 15. The hash partitioning distributes the records as follows: keys 1-5 to partition 1, keys 6-10 to partition 2 and keys 11-15 to partition 3. If there are any keys that are not in the range of 1 to 15 then those records will be assigned to a random partition. The real-time datasets may have multiple datasets with the same key and also they will not have all the keys mentioned in the range partition, i.e., kwys from 1 to 15. So range partitioning also cannot distribute the records equally across the partitions. Even if we manage to generate keys accordingly and distribute the keys equally we cannot guarantee proper load distribution that is because we cannot say that each record is of the same size.

It is clear that hash partitioning will be able to distribute the load equally only if the keys are equally distributed and range partitioning could evenly distribute the load only if the keys in each range are distributed equally. But in the real world datasets the keys are often not equally distributed and it leads to uneven distribution. In some cases we cannot be sure that the load is balanced just by equally distributing the keys. Because the amount of time taken to process each record might vary. To understand it in detail, take Figure 2.9 as an example. For example, 2 records with json data does not need to be of the same length. For instance, json data with GPS samples for different vehicles may vary based on the distance covered by the vehicle. So, one record can have only 80 samples whereas other record can have 750
2.5. Data Skew Problem in Distributed Computing Frameworks

Figure 2.8: Hash Partitioning.

Figure 2.9: Range Partitioning.
samples. But they both will be considered as one individual record. Let’s take a look at Figure 2.10 to understand what happens what if we distribute records with different sizes only based on the number of records.

In Figure 2.10, there are four records with different number of samples and think we managed to distribute the records equally on two partitions using range partitioning. Then the first two records will end up on Partition 1 and next two records will be on partition two. You could see that now Partition 1 has 160 samples and Partition has only 60 samples. Due to the huge difference between the number of samples on the partitions, i.e., data skew it results in higher processing times.

Since the default partitioning techniques used in distributed computing frameworks is not the best technique to use these frameworks for ER process. We propose a load balancing approach to handle data skew in Chapter 3.

2.6 Summary

In this chapter, we discussed the concept of ER, steps involved in the ER process, challenges faced in the ER process, and distributed computation frameworks used for parallel computing.
3. Methodology

In this chapter, we introduce our load balancing strategy for Entity Resolution (ER) called *Sampling-Based Precise Block Slicer* (SBPBS) to handle data skew problem in ER that occurs in a distributed environment. In Section 3.1, we discuss global workflow of SBPBS. Then, we discuss the two main parts of our designed strategy. In Section 3.2, we discuss the sampling module which is used to explore the block distributions and calculate the average block size. In Section 3.3, the details regarding our approach to divide the oversized block and assign blocks evenly to the partitions will be discussed. In Section 3.4, we discuss how we achieve a better load distribution. In Section 3.5, we discuss the similarity score calculation and record-pair comparison.

3.1 Global Workflow of SBPBS

Figure 3.1 shows the global workflow of SBPBS. Below, we discuss in detail about each step involved in the workflow.

The workflow is divided into four main modules. They are *Sampling, Precise Block Slicer (PBS), Block Distribution*, and *Entity Comparison*. Sampling module is responsible for sampling the input dataset, generating block statistics and calculating average block size. PBS module takes care of dividing larger block based on the average block size calculated in the sampling module. Block distribution module is responsible for distributing the blocks equally across all the partitions to achieve a proper load distribution. Whereas entity comparison module performs the record-pair comparison to determine whether a record-pair is a match or non-match.

Before sampling, some preparation steps are required. First, read the input dataset and generate the blocking key for each record in the dataset. This blocking key is generated by taking one or more features from the dataset (discussed in Section 2.2.2 and this makes sure all the similar records get a same blocking key. This helps to create blocks by grouping all the similar records in order to reduce the search space. Once the blocking keys are generated on the input dataset it is then given to the sampling process to get block distribution statistics in the dataset.
3. Methodology

Figure 3.1: Sampling-Based Precise Block Slicer's Global Workflow

Data Source

Start

Read Input Dataset

Generate Blocking Keys

Calculate Average Comparisons
Calculate Total Comparisons
Sample Input Dataset

Sampling

Divide Larger Blocks into Sub-Blocks

Precise Block Slicer

Block Distribution

If BC > AC

No

Yes

Partition By Partition ID
Assign Partition with Least Comparisons as Partition ID
Current Partition as Partition ID

Generate Comparison Pairs
Calculate Similarity Scores
Compare Record-Pairs

Stop

Database

Result

Entity Comparison
BC -> Block Comparisons
AC -> Average Comparisons
PC -> Comparisons in Partition
3.2 Sampling

In this section, we discuss the sampling process in SBPBS. We also discuss the sampling methods we considered. We discuss these sampling methods below:

Random Sampling: Random sampling chooses a subset of samples randomly from a given set. Each sample in the set has the same probability to be chosen at any time in the process [42].

Reservoir Sampling: Reservoir sampling is similar to random sampling but it maintains a better distribution of samples compared to random sampling. It actually provides real randomness. This sampling method uses a fixed size of blocks called the reservoir. First, it fills the reservoir with first \( n \) blocks and then selects \( i + n^{th} \) block from the set of population and replaces a block randomly from the reservoir by iterating through all the blocks in the set of population. Hence we can achieve a better distribution of samples [43].

Stratified Sampling: This is an in-built sampling method in Spark [44]. Stratified sampling divides the whole population into sub-groups and then simple random sampling is applied to each sub-group [45]. In our case, this ensures that samples from each block are randomly selected. This method allows us to specify only the fraction of samples that need to samples from the whole set of the population, so we specify to get the exact number of desired samples.

Progressive sampling: This sampling method does not require a sampling ratio or the sample size to be set before starting the sampling process. It determines the size of the sample based on the quality of the sampled dataset. It actually performs the actual task beforehand on the sampled dataset to determine the data quality and stop the sampling process. Even though it is not required to specify the sample size it is possible to specify if required [46].

These are the four sampling methods we considered for using the sampling module. We discuss the steps involved in sampling module below:

1. Sampling Input Dataset and Getting Block Distribution Statistics: In this step, we collect the block distribution statistics. It contains the information about the blocking key and its size. This information is later used to calculate the threshold value for splitting larger blocks that could cause data skew. The main advantage of using sampling to get the block distribution statistics reduces a huge overhead on pre-processing and also reduces the overall processing time.

In order to obtain the block distribution statistics, we take only a part of the dataset with blocking key using a sampling method. Usually, the datasets used in ER are huge and processing the whole dataset takes a lot of time, so we take only a part of the dataset in order to reduce the processing time for obtaining the statistics. The sampled dataset is then used to predict the block distribution statistics for the whole dataset by grouping the records in the sampled dataset with the blocking key. It is stored as a key-value pair with blocking key as a key and the number of records with the same blocking key \( (N_{BK}) \) as the value. This value is used to predict the number of records for a blocking key \( (P_{BK}) \) in the whole dataset using Equation 3.1.

\[
P_{BK} = N_{BK} \times sampling\_ratio
\] (3.1)
3. Methodology

Once all the records for each blocking key (block size, $B_{\text{Size}}$) is obtained, we calculate the number of comparisons needs to be performed in each block for ER. We calculate the number of comparisons for each block ($C_{BK}$) using Equation 3.2 and then we store these values as key-value pairs. This will be used for calculating block's threshold.

\[
C_{BK} = \frac{B_{\text{Size}} \times (B_{\text{Size}} - 1)}{2}
\]  

(3.2)

2. Calculating Block’s Threshold: From the block distribution statistics obtained from the previous step, we calculate block's threshold limit to determine whether a block is larger and needs to split into smaller blocks. The threshold $t$ is actually the number of comparisons allowed in each block. It is calculated by dividing the total number of comparisons ($C_{\text{Total}}$) with the total number of blocks ($B_{\text{Total}}$) obtained from block size statistics as shown in Equation 3.3. Even though load distribution depends on the threshold value, it is not that sensitive to be affected by the difference between the actual threshold value and the predicted value. Therefore, sampling is feasible to be used, even though its results may not be 100% accurate.

\[
C_{BK} = \frac{C_{\text{Total}}}{B_{\text{Total}}}
\]  

(3.3)

3.3 Precise Block Slicer

In this section, we discuss the Precise Block Slicer (PBS) strategy to divide the number of comparisons in the larger blocks into sub-blocks with comparisons equal to the threshold value.

This module is the heart of SBPBS, once the threshold value for the block is calculated, we get the original input dataset with blocking key and group them to form blocks. Then, we use the number of comparisons for each blocking key information obtained earlier and the threshold $t$ to split the larger blocks into sub-blocks. It is done recursively until the final sub-block with the number of comparisons less than $t$ is achieved. We split the blocks in such a way that the number of comparisons in every sub-block except the final sub-block has the exact same number of comparisons as the threshold value. We call this process of splitting the blocks as Precise Slicing. The concept of Precise Slicing is explained in Algorithm 1:
Precise Block Slicer Algorithm

Input
blocks - grouped records with blocking, threshold - maximum number of comparisons allowed per block.

**Algorithm 1:** Precise Block Slicer Algorithm

1. \( \text{bsize} = \text{Block Size}; \)
2. \( \text{comparisons} = \text{bsize} \times (\text{bsize} - 1) / 2; \)
3. \( \text{avgComparisons} = \text{threshold}; \)
4. \( \text{extraComparisons} = 0; \) // comparisons from previous iteration
5. \( \text{comparisonsCount} = 0; \) // comparisons in a sub-block
6. \( \text{topRecords} = 0; \) // first n records from a list chosen for generating comparison pairs
7. \( \text{remainingSplitRecords} = 0; \) // records moved to next iteration due to precise slicing
8. \( \text{comparisonsToNextBlock} = 0; \) // number of comparisons moved to next iteration
9. \( \text{preciseSplitting} = \text{true}; \)
10. \( \text{startIndex} = 0; \)
11. \( \text{if} \ \text{extraComparisons} > 0 \ \text{then} \)
12. \( \quad \text{topRecords} = 1; \)
13. \( \quad \text{startIndex} = 1; \)
14. \( \text{end} \)
15. \( \text{for} \ i = \text{startIndex}; \ i < \text{bsize}; \ i + + \ \text{do} \)
16. \( \quad \text{if} \ \text{comparisonsCount} + (\text{bsize} - i - 1) \leq \text{avgComparisons} \text{ then} \)
17. \( \quad \quad \text{topRecords}++; \)
18. \( \quad \quad \text{comparisonsCount} += (\text{bsize} - i - 1); \)
19. \( \quad \text{end} \)
20. \( \text{else if} \ \text{preciseSplitting} == \text{true} \text{ then} \)
21. \( \quad \text{if} \ \text{comparisonsCount} \neq \text{avgComparisons} \text{ then} \)
22. \( \quad \quad \text{topRecords} = \text{topRecords} + 1; \)
23. \( \quad \quad \text{remainingSplitRecords} = \text{avgComparisons} - \text{comparisonsCount}; \)
24. \( \quad \quad \text{comparisonsCount} = \text{comparisonsCount} + \text{remainingSplitRecords}; \)
25. \( \quad \quad \text{comparisonsToNextBlock} = (\text{bsize} - i - 1) - \text{remainingSplitRecords}; \)
26. \( \quad \text{end} \)
27. \( \quad \text{preciseSplitting} = \text{false}; \)
28. \( \text{end} \)
29. \( \text{else} \)
30. \( \quad \text{break}; \)
31. \( \text{end} \)
32. \( \text{end} \)

Algorithm 1 explains how we perform precise slicing. Blocks and the threshold value are taken as the input. Lines 1 to 10 shows the variables used in the algorithm. Lines 11 to 14 takes care of setting the starting index of the sub-block. It sets the start index to 1 if there are some comparisons that need to be added from a previous sub-block into this sub-block. Lines 15 to 32 are responsible for precise slicing. Lines 16 to 19 add a new record to be considered for creating record-pairs if the total number of comparisons in this sub-block will still be less than the average comparisons even if we generate comparison pairs for the current record. Or else, it is given to the
Total comparisons = \( \frac{\text{Total Records} \times (\text{Total Records} - 1)}{2} \)

Total Comparisons = \( 7 \times 6 / 2 = 21 \)

Assume, Threshold = 8

Figure 3.2: Concept of Precise Block Slicer
else if condition which is from line 20 to 28. It checks whether this block should be considered for precise slicing. If yes, it then sets remainingSplitRecords to the number of comparisons that could be still be allowed in this sub-block and it also sets the value for comparisonsToNextBlock, this is the number of comparisons that should be moved to the next sub-block for the record which we applied precise slicing. It also updates the total comparisons count. Then it sets the preciseSplitting variable to false to make sure that we will not perform precise slicing anymore for this block. In the next iteration both the if conditions in line 16 and line 20 will not be satisfied as we have already reached the threshold limit and we change preciseSplitting to false in line 27. Then the algorithm moves to the next sub-block and it continues until it gets a sub-block with comparisons less than the threshold value.

To understand PBS in detail let’s take a look at Figure 3.2 which explains how precise block slicer works. As shown in the figure, there is a block with 7 records and 21 comparisons need to be performed in total. Let’s assume the threshold as 8 comparisons per block. Since precise block slicer allows only the exact number of comparisons as the threshold value in a block, it will result in 3 blocks. In precise block slicer, all the sub-blocks except the last sub-block has the same number of comparisons i.e., the threshold value. Since the number of comparisons in the last sub-block are already less than the threshold value that sub-block is left as it is. This guarantees that there will not be any block with the number of comparisons more than the threshold value and it will make the load be more balanced. The next step after slicing the blocks is to generate record pairs and compare them.

Let us breakdown Figure 3.2 by each sub-block and understand what happens in each iteration below:

1. **First Iteration:** In the first iteration there are 7 records in the block which would normally generate 21 record-pairs but based on the threshold $t$ we should allow only 8 comparisons per block/sub-block. So, we give the block information to PBS algorithm Algorithm 1. The block information includes the size of the block which is 7, threshold $t$ which is 8 and extra comparisons i.e., the left over comparisons for a record from the previous block, but here since it is the first sub-block it will be zero. Once these values are given to the PBS algorithm, it calculates the number of records from the block those will be compared with other records and we call it as toprecords. In this case, we only compare A with all other records which take 6 comparisons. So it sets the value of toprecords to 1. Now, only 2 comparisons can be performed in this sub-block. Now it increments the toprecords since we have taken the second record in the block which is B for generating record-pairs. The PBS algorithm returns three values: toprecords, remainingSplitRecords, comparisonsToNextBlock: remainingSplitRecords are the number of record-pairs we generate for the last toprecord which shows the precise slicing, in our case the value for the first iteration will be 2. comparisonsToNextBlock is the value which say that how many record-pairs from the last toprecords are moved to the next sub-block, here it will be 3 as we move three record pairs ((B,E), (B,F), (B,G)) to next sub-block.

2. **Second Iteration:** In this iteration, we will have a block size of six. Since the PBS algorithm returned 2 toprecords and also 3 comparisonsToNextBlock, we know that we still need the last record of the toprecords from the previous sub-block. We always
subtract one from the toprecords if comparisonsToNextBlock is not zero. The number of comparisons required at this stage is 13 since we already performed 8. So we need to again do precise slicing. In this iteration, we give three values to PBS algorithm like the same way as we did in the previous iteration but this time we have extra comparisons. So the values for block size, threshold and, extra comparisons will be 6, 8 and 3 respectively. Now the PBS algorithm has a value for extra comparisons, so it subtracts the extra comparisons from the threshold value and performs precise slicing. Here three records were taken for generating records-pairs which are B, C, D. So then the value for toprecords will be 3. Since for record D, only record-pair is generated remainingSplitRecords will be 1 and comparisonsToNextBlock will be 2 as we still need to generate record-pairs (D,F) and (D,G).

3. Third Iteration: Now, we are required to generate only 5 comparisons as shown in the last sub-block (Figure 3.2) which is less than the threshold value that does not need to be split into a sub-block. So that will be the last sub-block of the larger block which had 7 records with 21 comparisons.

3.4 Block Distribution

In this section, we discuss how we distribute the blocks and sub-blocks generated in Section 3.3 across all the partitions.

Block distribution module is responsible for assigning the partition ids to the blocks in order to distribute the blocks equally across the partitions. The block size is calculated based on the number of comparisons needs to be performed in that particular block. We calculate the number of comparisons that can be allowed on each partition by dividing the total number of comparisons in the whole dataset calculated in Section 3.2 by the total number of partitions. We store the information about the number of comparisons on each partition in a collection object which we call as partition pool.

For each block the total number of comparisons are checked and it looks into the partition pool whether the block can be kept on the same partition where it already is. If the sum of already existing comparisons on that partition and the number of comparisons in the block are less than or equal to the number of comparisons allowed on that particular partition, then that block will be given an ID as the partition number. Else, it checks the partition that has the less number of comparisons so far by looking into the partition pool and assigns that partition number as ID. Finally, the blocks are re-partitioned based on their IDs. This makes sure the blocks are equally distributed across the partitions.

3.5 Entity Comparison

In this section, we discuss how we generate record-pairs and how we compare them based on the obtained similarity scores.

This module performs the entity comparison. After the blocks are split into sub-blocks, the records in each block are combined to generate record-pairs. These record-pairs will be generated by pairing each record with every other record in the
block and the same pair of records are not repeated twice. For example, if records A and B are created as a pair (A,B) we don’t need to create a pair of (B,A). The number record-pairs (comparisons) in each block or sub-block can be calculated using Equation 3.2. Once the record-pairs are generated they are compared using character based similarity techniques explained in Section 2.2.3. After comparison, first similarity scores are calculated for each attribute using proper similarity functions, then a total similarity score is obtained by summing up all similarity scores. Based on the total similarity score, a threshold-based classification is used to classify each pair to match and non-match pairs. This classification is done by techniques discussed in Section 2.2.4. Finally, after classification the results can stored in a database or can be used for further analysis.

3.6 Summary

In this section, we introduced our proposed load balancing for the ER approach called Sampling-Based Precise Block Slicer and how it works. In Chapter 4, we discussed how we implemented this approach using Apache Spark.
4. Implementation of Sampling-Based Precise Block Slicer Using Apache Spark

In Chapter 3, we discussed Sampling-Based Precise Block Slicer (SBPBS) and how it works. In this chapter, we will discuss the implementation of SBPBS using Apache Spark. Based on the distributed computing frameworks discussed in Section 2.4, we have chosen Apache Spark for implementing our load balancing approach for Entity Resolution (ER). The reason for choosing Apache Spark is its faster computation, fault-tolerance, stability and its easy integration with Hadoop Distributed File System (HDFS) [47]. In Section 4.1, we discuss the architecture of SBPBS and its workflow. In Section 4.2, we discuss the spark stages and actions in SBPBS's Spark implementation.

4.1 Block Diagram of SBPBS

Figure 4.1 shows the architecture of SBPBS. It is divided into four modules. They are: sampling, precise block slicer, block distribution, and entity comparison. Let’s discuss these modules in the following sections.
4.1. Block Diagram of SBPBS

Figure 4.1: Block Diagram of SBPBS
Before we start sampling the input dataset, we perform some required steps. The first step is to read the input dataset and generate the blocking keys for every record in the dataset. The records are then grouped into blocks based on the generated blocking key in order to reduce the search space. The standard blocking technique discussed in Section 2.2.2 is used to generate blocks.

In our implementation, we used a dataset with personal information of people that contains fields like givenname, surname, address, telephone-number and so on (details are given in Section 5.1.1). We generate the blocking key using Double Metaphone technique discussed in 3, this function takes one or more column names. We have taken two columns namely given-name and surname separately as doubleMetaphone (given-name), doubleMetaphone(surname) and it returns a value for both of them. Let’s assume the values to be $gn_{dm}$ and $sn_{dm}$ for given name and surname respectively.

Then, the first two letters of $gn_{dm}$ and $sn_{dm}$ are taken and concatenated to create a blocking key: $gn_{dm}$ and $sn_{dm}$. This blocking key is added as a new column and it is used to group all the similar records based on the blocking key in the following steps.

Figure 4.2 shows the table with givename,d, surname,d and the blocking key generated for the records in the input dataset.

![Table](image)

Figure 4.2: Example of Records with Blocking Key

### 4.1.1 Sampling

This module takes care of sampling the dataset, collecting block distribution statistics and predicting the total number of comparisons in the dataset. This will be used to calculate threshold $t$, $t$ is later used for dividing the larger blocks into sub-blocks.

As we discussed in Chapter 3, We implemented three of the sampling methods discussed in Section 3.2. They are Random sampling, reservoir sampling, and stratified sampling. Considering the time factor, we couldn’t implement progressive sampling because of the high complexity in setting the stopping condition. After testing all the three sampling methods we used stratified sampling in our approach because of its speed and less memory consumption for implementation as it is an in-built sampling method in Spark. We sample the dataset in order to reduce the pre-processing time required to collect the block distribution statistics. Here we sample the dataset and try to predict the block distribution statistics for the whole dataset. We tested different sampling ratios and analyzed the difference in performance (discussed in Section 5.2).
We use HashMap to store block distribution statistics. As discussed in Section 4.1, once the records in the sampled dataset are grouped to create blocks. The size of each block is calculated and stored in the HashMap as key-value pairs with blocking key as the key and the size of the block as the value. Then, Equation 3.2 is applied on all the values to calculate the number of comparisons for each block. This HashMap will be used in Precise Block Slicer to generate keys for each record. Since we collect only the information about blocking keys and not any complicated information like partition number or node ids. So we could able to store them in a simple HashMap as key-value pairs. Storing them in HashMap will also help us to have a quicker look up while checking the number of comparisons for each block (in Stage 2 in Figure 4.6). This also takes up very less memory which is more important while working with in-memory computing frameworks like Spark.

Calculating the Threshold Value $t$: In order to calculate $t$, we use Equation 3.3 where we take the sum of all the values from the HashMap we created earlier and dividing it by the total number of keys in the HashMap. In other words, we could say dividing the total number of predicted comparisons by the total number of blocks from the sampled dataset. Once $t$ is calculated we can take the original input dataset and check whether there are oversized blocks.

4.1.2 Precise Block Slicer

This module is the heart of our approach to handling the data skew problem in the ER. In this section we will discuss how we used Apache Spark to implement Precise Block Slicer (PBS) which is explained in Section 3.3. The steps involved in implementing PBS and how the key (composite-key) will be at every stage are explained below:

1. **Group Records By Blocking Keys:** Once block statistics are obtained all the records in the input dataset with blocking key are grouped based on the blocking key. At the end of this step, the key will be the same as the blocking-key, for example, LMOD.

2. **Divide Larger Blocks into Sub-Blocks:** Every block with the number of comparisons more than the threshold value is divided into sub-blocks and the maximum number of comparisons allowed per sub-block will be exactly the threshold value. The number of comparisons is obtained from the HashMap with the number of comparisons for each blocking key created in Section 4.1.1. The larger block or a sub-block is considered as a list of records and we divide it if the number of comparisons in that block/sub-block exceeds than the threshold using PBS discussed in Chapter 3. The algorithm will return the number of top records from the list that should be taken for comparisons, the starting index of the first record for comparison, the stopping index of the last record of top records and the number of comparisons moved to the next sub-block. Figure 4.3 shows how the larger records are split and how starting and stopping indexes are generated. This information is used to create a composite key for generating record-pairs for comparison. The composite key will consist of the blocking-key and number of comparisons in that block, for example, LMOD.21.

Figure 4.3 shows an example how Precise Block Slicer(PBS) works. Let’s consider we have a block with 7 records and assume threshold as 8. We need to perform 21
comparisons, so the size of the list is 7, threshold 8 is given to the PBS function and it returns the number of top records to be taken for making record-pairs which are 2, and the start index of the first record for the next sub-block which is 3, and the comparisons which are being moved to next sub-sub-block i.e., extra comparisons three. We could see for the next sub-block we send the extra comparisons to the PBS function so it subtracts the extra comparisons from its threshold value and returns top records as 1, start index for next sub-block as two since it has two extra comparisons. We could also see the leftover comparisons (D,F) and (D,G). We could see the start index set in the previous set block is used in the second sub-block where the comparison started from the third element that is E. Now the same happens for the third sub-block. The number of comparisons in the third sub-block is less than the threshold value and this block will not be further divided into a sub-block.

3. Generate Composite Key (Start and Stop Indexes for sub-blocks):

Once the PBS creates a new sub-block if the number of comparisons in the block exceeds the threshold value, a composite key is generated for every block/sub-block. The number of top records (obtained in the previous step) that needs to be compared with all the other remaining records is added to the composite key. Usually, when the blocks have split the records that need to be compared for a record at index i will be i+1, i+2, ..., n. This results in uneven sub-block sizes. Since in this approach the maximum number of comparisons in a sub-block except the last sub-block has exactly the number of comparisons as the threshold value, the starting index of the record that needs to be compared for the first record might not be i+1 and also the last record that needs to be compared for the final record that is taken for comparisons doesn’t need to be nth record. So, the starting index of the first record and the stopping index of the last record of top records (from the previous step) is added to the composite key along with the blocking key. For example, it will look like LMOD.5.2.4 where 5 is the number iteration to be performed, i.e., the number of records that should be taken for generating record-pairs with other records in the block. The iteration starts from the first records. Whereas 2 defines the start index of the first iteration when the first record is taken for generating record-pairs it checks the start index to know from which index of the list the first record should start generating record-pairs. The stop index which is 4 in the example composite key determines how many record-pairs are allowed for the last record in the iteration.

4.1.3 Block Distribution

This module takes care of the distribution of block across all the partitions in order to achieve better load balancing. The steps performed in this module are discussed below:

1. Generating Partition ID as Key: After the composite key is created for the sub-blocks in the last step of PBS module (Section 4.1.2), they are assigned with partition ids. An array with the size of the number of partitions is created and filled with the average number of comparisons. This array is used as a partition pool to track the partitions with the least number of comparisons allocated. While assigning a Partition ID for each block, first the current partition id of the block is obtained
4.1. Block Diagram of SBPBS

Figure 4.3: Splitting Larger Blocks Using PBS

Comparisons = 21
Assume, Threshold = 8
Initially, extraComparisons = 0
startIndex = 0;

removeRecords = 1;
startIndex = 3;
extraComparisons = 3

Figure 4.3: Splitting Larger Blocks Using PBS
and the number of comparisons allowed for that partition is checked in the partition pool. If the number of comparisons allowed for that partition is greater than the number of comparisons for that particular block/sub-block then the partition ID is set as the same the current partition ID else the partition with more number of comparisons allowed is set as the partition ID. The output of this step will be JavaPairRDD. The key will be the partition ID and the value will be the output from the previous step which had a key-value pair (key-composite, value-record) will be stored in Tuple2. The records are partitioned based on the key(partition ID) of the JavaPairRDD.

This distributes the total comparisons among all the available partitions evenly and instead of moving all the blocks it tries to hold the block/sub-block in the same partition if the number of comparisons in that block/sub-block doesn’t exceed the average number of comparisons for that partition thus reducing the shuffling of records among different nodes in the cluster. This reduces the network I/O on the cluster and results in increased overall efficiency.

2. Re-Partitioning Based on Partition ID: Once the partition IDs are assigned for all the blocks/sub-block they are re-partitioned to achieve a balanced load (better parallelism) which results in reduced processing time. This usually causes a huge amount of shuffling between the partitions but because of the Partition Pool strategy used in the previous step the amount of shuffling is reduced and almost equal load distribution is achieved.

4.1.4 Entity Comparison

This is the last final stage of the implementation where ER is performed, this module takes care of generating record pairs for comparison and evaluating them. The steps involved in this module are explained below:

1. Generating Record Pairs: Here, the pair of records those needs to be compared are combined together. Now, we have already created blocks with the number of comparisons less than the threshold value. The composite key we generated in the previous step is added as a key to the block of similar records whereas the block will be as the value of the key-value tuple. The composite key consists of start and stop index which is used to iterate through the sub-blocks to create record pairs.

Once the records are grouped into blocks based on blocking-key, each record in the block only needs to be compared with the records within the block. So the comparison record pairs are generated accordingly and the comparisons are performed in the next stage.

2. Comparing the Record-Pairs: Once the record pairs are generated they are compared and a comparison score is given to every record pair. For each pair of record, two columns of the same attribute from both records are compared. For example, when there is a record pair with record A and record B, the column “surname” of record A will be compared against “surname” of record B and this for all the remaining columns.

The comparisons for columns with string values Jaro Winkler (discussed in Section 2.2.3) is used. Jaro Winkler gives an output value between 0 and 1. The
4.2 Spark Stages and Actions of SBPBS

closer the value is to 1, more the similarity between the two records. Whereas for the columns with numeric values the absolute values are compared. If the similarity value is 1 then the records are similar and if the similarity value is 0 then the records are non-similar.

![Figure 4.4: Records with similarity values.](image)

Figure 4.4 shows a table with record-pairs and their comparison scores for columns given_name, surname, city, and age. The comparison scores are rounded off to four decimal values. This score is then used to determine whether the record pair is a match and then the results are stored in HDFS.

![Figure 4.5: Classification step sample output.](image)

Figure 4.5 shows the total similarity sum and label column that says 'match' or 'nonmatch'. Once the comparison pairs are classified the results are saved to HDFS.

3. Classification: This is the final step of the entity comparison module. Here each comparison-pair is determined as ‘match’ or ‘non-match’ using threshold-based classification model (discussed in Section 2.5). This model is based on a numerical threshold value t. In the previous step, comparison-pair similarity metrics are calculated and the sum of these metrics is called ‘total_similarity_sum’. The total_similarity_sum value is compared against threshold value t to determine whether the comparison pair is ‘match’ or ‘nonmatch’. For instance, let’s consider a comparison pair (rec1, rec2) and the threshold value t be 7. Figure 4.5 shows the total_similarity_sum and label column that says ‘match’ or ‘nonmatch’. Once the comparison pairs are classified the results are saved to HDFS.

4.2 Spark Stages and Actions of SBPBS

Figure 4.6 shows the table with actions and stages in SBPBS. There are five stages including two actions. The first action collectAsMap (stage 1) triggers the stage mapToPair (stage 0). This mapToPair and collectAsMap are responsible for calculating the total number of predicted comparisons for each block and storing them in a HashMap.
4.3 Summary

In this chapter, we discussed how we implemented SBPBS using Apache Spark. We will discuss the evaluation of SBPBS load balancing approach with other ER load balancing approaches in Chapter 5.
5. Evaluation

In this chapter, we evaluate Sampling-Based Precise Block Slicer (SBPBS) based on four different metrics. They are execution time, speedup, scalability, robustness, and degree of parallelism. In Section 5.1, we take a look at the experimental setup used for evaluation along with the datasets used and, the cluster specifications (both software and hardware). In Section 5.2, we discuss and evaluate the benefits of using sampling and also results obtained for different sampling methods. In Section 5.3, we discuss choosing the optimum values for spark parameters for running spark jobs. Section 5.4, we discuss the experiments performed to evaluate the other ER load balancing approaches from the aforementioned five metrics.

5.1 Experimental Setup

In this section, we discuss the testing environment used for evaluating our load balancing approach based on different factors. First, we will start with the datasets used for evaluation (Section 5.1.1), then the specifications of the cluster (Section 5.1.2) and finally how to submit a spark job on the cluster (Section 5.1.3).

5.1.1 Input Datasets

In the evaluation process, we used an open source data generator called FEBRL [48]. Peter Christen developed a python based application as part of FEBRL project. The FEBRL data generator is used to create synthetic datasets because of the difficulties in acquiring gold standard data with varying skewed data and varying dataset sizes. The FEBRL data generator allows us to generate the datasets with the desired size and the amount of skewed data. Table 5.1 provides the details about the datasets used for evaluation.

Dataset Schema: FEBRL generates datasets with comma-separated values (CSV) containing personal information with the following fields: rec-id, given-name, surname, postcode, city, telephone-number, credit-card-number, income-normal, age-uniform, income, age, sex, blood-pressure.
Datasets | Total Records | Duplicates (%) | Size in MB
---|---|---|---
DS1 | \(1.05 \times 10^6\) | \(5 \times 10^4\) (5%) | 120
DS2 | \(1.1 \times 10^6\) | \(1 \times 10^5\) (10%) | 126
DS3 | \(1.2 \times 10^6\) | \(2 \times 10^5\) (20%) | 136
DS4 | \(1.3 \times 10^6\) | \(3 \times 10^5\) (40%) | 147
DS5 | \(1.4 \times 10^6\) | \(4 \times 10^5\) (40%) | 157
DS6 | \(1.5 \times 10^6\) | \(5 \times 10^5\) (50%) | 168
DS7 | \(1.7 \times 10^6\) | \(7 \times 10^5\) (70%) | 189
DS8 | \(2 \times 10^6\) | \(1 \times 10^6\) (100%) | 220
DS9 | \(3.3 \times 10^6\) | \(2.3 \times 10^6\) (70%) | 367
DS10 | \(3.5 \times 10^6\) | \(2.5 \times 10^6\) (75%) | 388
DS11 | \(5 \times 10^5\) | \(5 \times 10^4\) (10%) | 63

Table 5.1: Test datasets with various sizes used for evaluation.

<table>
<thead>
<tr>
<th>Node</th>
<th>Role</th>
<th>CPU</th>
<th>Cores</th>
<th>RAM(GB)</th>
<th>Disk(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Management</td>
<td>E5620@ 2.40GHz</td>
<td>4</td>
<td>16</td>
<td>120</td>
</tr>
<tr>
<td>2-3</td>
<td>NameNodes</td>
<td>E5620@ 2.40GHz</td>
<td>6</td>
<td>32</td>
<td>120</td>
</tr>
<tr>
<td>4-7</td>
<td>Executors</td>
<td>E5-2650@ 2.00GHz</td>
<td>4</td>
<td>16</td>
<td>150</td>
</tr>
<tr>
<td>8-9</td>
<td>Executors</td>
<td>E5-2650 v2@ 2.60GHz</td>
<td>4</td>
<td>16</td>
<td>150</td>
</tr>
<tr>
<td>10</td>
<td>Executors</td>
<td>E5520@ 2.27GHz</td>
<td>4</td>
<td>16</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 5.2: Cluster Specification.

### 5.1.2 Cluster Specifications

In order to test our application, we need a multi-node spark cluster with HDFS. So we used Hortonworks Data Platform (HDP) which is an open source distributed storage and data processing framework [49]. HDP comes with a suite of tools from Hadoop ecosystem that includes Apache Spark, YARN, MapReduce, HDFS, HBase etc. For our evaluation, HDP version 2.6.4.0-91 was installed on a heterogeneous cluster of 10 nodes running centos 7. All of the nodes are connected in a star topology via a 10 GBit Ethernet. Table 5.2 shows the specifications of the CPUs, number of cores, memory and disk size on all the nodes.

### 5.1.3 Submitting Spark Job to the Cluster

When we run the spark job in a cluster environment, we need to run on YARN. In this section, we discuss how to bundle the spark application and run it on the cluster.

First, we build our spark application using Maven\(^1\) build tool as a jar file with dependencies required for running the spark job. Then, to run the jar file we use `spark-submit` script with some additional parameters. These parameters are mentioned below:

- `--class`: path to the main class of the spark project.

\(^1\)https://maven.apache.org/
5.2 Evaluation of Sampling for Pre-Processing

- **–master:** cluster manager such as standalone, yarn, mesos or kubernetes. We use yarn.
- **–deploy mode:** set the spark job to run on the worker nodes (cluster) or on an external node (client). We use cluster.
- **num-executors:** number of executor instances that should be created. In case of insufficient resources such as less cores or less memory only the maximum possible executors are created.
- **–executor-memory:** memory allocated for each executor.
- **–executor-cores:** number of processor cores assigned for each executor.
- **driver-cores:** number of processor cores assigned to driver.
- **–driver-memory:** memory allocated to the driver instance.
- Finally the path of the spark application jar file and the arguments for the jar file are given.

The values for the above-mentioned parameters vary based on the available resources and size of the input dataset.

**Example spark-submit:** spark-submit –class ER_LoadBalancing.SBPBC –master yarn –deploy-mode cluster –driver-cores 1 –driver-memory 1g –executor-cores 4 –num-executors 5 –executor-memory 6g entity-resolution-1.0-SNAPSHOT-jar-with-dependencies.jar 20 1 1 0.25 4

5.2 Evaluation of Sampling for Pre-Processing

In this section, we discuss the sampling methods we considered and why we chose stratified sampling. In Section 5.2.1, we discuss the sampling methods evaluated for using in SBPBS. In Section 5.2.2, we discuss the impact of sampling percentage on the processing time of SBPBS. We will also compare the sampling method with BDM in Section 5.2.3.

5.2.1 Evaluation of Sampling Methods

In this section, we discuss three sampling methods we evaluated. They are random sampling, reservoir sampling, and stratified sampling. The evaluation of these sampling methods is shown in Table 5.3. We used random sampling without replacement which means that we don’t pick the same record twice. We evaluated these three sampling methods by varying sampling percentages at 20% intervals using a dataset with 105000 records and 5000 duplicate records.

From the results in Table 5.3, we see that reservoir sampling performs slightly better than the other two sampling methods when the sampling percentage is less. For 100% sampling, stratified sampling has the threshold value as 346 even with all the samples when other two had 345 is because we specify the fraction, not the sample
size (count), so there might be a slight difference even if we specify 1 (fractional value for 100%) [44]. All the three sampling methods have almost equal prediction value with higher percentages. Because the fraction we specify will get an approximate number of samples. This slight difference in predicted threshold value does not affect the processing time (discussed in Section 5.2.2). Also when we implemented these three sampling methods in SBPBS, we have to use collect on the input dataset which brings a huge amount of data to the driver and causes the driver to fail when it is used for large datasets. But this does not happen with stratified sampling because its in-built sampling method in Spark and we don’t need to perform the action \textit{collect} on the input dataset as the sampling is taken care by Spark. Stratified sampling will also be faster than the other two sampling methods for the same reason. In order to test that, we calculated the time taken by these three methods for various sampling percentages. The results were shown in Table 5.4. We could see that the change in the sampling percentage does not have a huge impact on the sampling time of stratifies, unlike other two sampling methods. So, we used stratified sampling in SBPBS to perform all the evaluations against other ER approaches which will be discussed in the coming sections.

### 5.2.2 Impact of Sampling Percentage on Processing Time

In this section, we evaluate the benefits of using sampling and its effect on the threshold value to split the block (discussed in Chapter 3).

Since the threshold value is used to split the larger blocks, there will obviously be a slight difference in the threshold value for different sampling percentages. Since the threshold value has an impact on load distribution, we also tested that a slight difference in threshold value does not make a huge difference in processing time, we evaluated the processing time of SBPBS for different sampling percentages. For evaluation, we used stratified sampling and dataset DS2 (given in Table 5.1) with

<table>
<thead>
<tr>
<th>Sampling Percentage</th>
<th>Stratified</th>
<th>Reservoir</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td>660</td>
<td>646</td>
<td>663</td>
</tr>
<tr>
<td>40%</td>
<td>473</td>
<td>464</td>
<td>477</td>
</tr>
<tr>
<td>60%</td>
<td>408</td>
<td>405</td>
<td>400</td>
</tr>
<tr>
<td>80%</td>
<td>366</td>
<td>367</td>
<td>367</td>
</tr>
<tr>
<td>100%</td>
<td>346</td>
<td>345</td>
<td>345</td>
</tr>
</tbody>
</table>

Table 5.3: Predicted threshold value.

<table>
<thead>
<tr>
<th>Sampling Percentage</th>
<th>Stratified</th>
<th>Reservoir</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td>8</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>40%</td>
<td>9</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>60%</td>
<td>10</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>80%</td>
<td>10</td>
<td>42</td>
<td>39</td>
</tr>
<tr>
<td>100%</td>
<td>11</td>
<td>181</td>
<td>178</td>
</tr>
</tbody>
</table>

Table 5.4: Sampling time.
5.2. Evaluation of Sampling for Pre-Processing

<table>
<thead>
<tr>
<th>Sampling Percentage</th>
<th>Processing Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td>290</td>
</tr>
<tr>
<td>40%</td>
<td>295</td>
</tr>
<tr>
<td>60%</td>
<td>292</td>
</tr>
<tr>
<td>80%</td>
<td>296</td>
</tr>
<tr>
<td>100%</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 5.5: Evaluation of SBPBS's processing time for different sampling percentages.

different sampling percentages: 20%, 40%, 60%, 80%, and 100%. The processing times for the aforementioned sampling percentages are given in Table 5.5.

The processing time for various sampling percentages in Table 5.5 shows the difference in sampling percentages, which in turn affects the predicted threshold value, does not impact much the overall processing time but meanwhile we could also see that the processing time increases slightly because of the time taken for sampling the input dataset. This shows that we don’t need to use a high sampling percentage rather we could go with a smaller sampling percentage which will reduce the overall processing time significantly while processing larger datasets. It also shows that choosing the sampling percentage is a trade-off as we need to choose a sampling percentage to get a good threshold value but also should not take more time for preprocessing. Because we can see from the results that even with 100% sampling which will give a precise threshold value the processing time is more than the processing time with 20%.

5.2.3 Comparison Between Stratified Sampling and Block distribution Matrix (BDM)

In this section, we compare and evaluate the difference between using stratified sampling and BDM for collecting block distribution statistics. BDM is used by state-of-the-art ER load balancing approaches namely Block Slicer and Block Split (discussed in Section 6.3.1) to store block distribution statistics. It uses this special data structure to store blocks information in the form of a matrix. It also stores partition and node information of each block. This makes the BDM huge and takes time to store and retrieve information from BDM. It will also become a limitation when the size of the dataset increases. Where when we use stratified sampling for collecting block information, we get only the blocking-key and the block size. We are not getting any information about the partitions or nodes. We then store the blocking-keys and block sizes in a HashMap that does not take much memory and also it will be faster for lookup.

In order to test this, we calculated the time taken to collect block distribution statistics using BDM and stratified sampling. Since we want to evaluate both the methods based on block information collection, we used the sampling percentage for stratified sampling as 100%. Such that both the methods will have the same number of input records. We used DS1 (given in Table 5.1) for evaluation. When we tested both the approaches, BDM took 35 seconds to collect block statistics whereas collecting block information using stratified sampling took only 20 seconds.
This shows that collecting only minimal information (blocking-key and block size) about the blocks and storing it in a HashMap are much faster than using a BDM. This will also help us to retrieve the size of the blocking keys faster in the later steps where we get the size of each blocking key to check whether the size of the block is less than the threshold value.

5.3 Tuning Spark Parameters

In this section, we discuss tuning spark parameters in order to achieve better parallel processing. As executors being the core of data processing time, optimizing executor parameters has a huge influence on achieving better data processing. Driver optimization should be done based on the driver program. We can achieve the best parallelization only when we make use of all the available processor cores in the cluster [50].

We have spark installed on 7 nodes with 4 cores each which makes a total of 28 cores. Since each input partition will be assigned to each core, we need to partition the input dataset with at least 28 partitions. If the number of partitions is less than the number of the available cores, then only the same number of cores as the number of partitions will be used and the remaining will be idle. For example, we now have 28 cores and if we partition the input dataset into 20 partitions, only 20 cores are used for processing the dataset and the remaining 8 cores will be idle. So it is recommended to have the input partitions as multiples of the number of cores such as 28, 56, 84 and so on. Since increasing the number of partitions increases data shuffling we need to choose the optimal number of partitions. In the below sections, we discuss how we have chosen an optimum number of partitions for all the aforementioned four ER approaches using DS2 (shown in Table 5.1).

5.3.1 Optimum Partitions - Basic Approach

We used the dataset DS2 (shown in Table 5.1) to find the optimum number of partitions for the basic approach (discussed in Section 2.2). We ran the basic approach spark job with partitions 28, 56, 84, 112, 140 and then chosen the partitions with least processing time as the optimum partitions. Figure 5.1 shows a graph plotted between the number of partitions and processing time. Based on the results as shown in the graph we chose 56 partitions as the optimum number of partitions for Basic approach.

5.3.2 Optimum Partitions - Block Split

We do the same as we did with the basic approach to find the optimum number of partitions. Figure 5.2 shows the graph drawn between the number of partitions and Processing Time for the corresponding number of partitions. Based on the experiment, we chose the optimum number of partitions as 56.

5.3.3 Optimum Partitions - SBPBS

The optimum number of partitions is chosen in SBPBS the same as Basic approach and Block Split. The sampling percentage uses is 25%. Figure 5.3 shows the graph of
5.3. Tuning Spark Parameters

Figure 5.1: Processing time for different number of partitions - Basic Approach

Figure 5.2: Processing time for different number of partitions - Block Split
the processing time taken by SBPBS to process DS2 (shown in Table 5.1) dataset for a different number of partitions. This experiment shows that the difference between the processing times is very less and we have chosen 28 as the optimum number of partitions for SBPBS.

![SBPBS Graph](image)

Figure 5.3: Processing time for different number of partitions - SBPBS

5.3.4 Optimum Partitions - Block Slicer

We find the optimum number of partitions in the same way as we did for the previous three approaches. Figure 5.4 shows the graph for the time taken for processing DS2 (shown in Table 5.1) using Block Slicer approach dataset for a different number of partitions. Based on the results obtained we have chosen 28 partitions.

Once the number of partitions is chosen we need to select the parameters for the driver. We have given 1GB memory and 1 core to the driver as it is not doing the heavy lifting. We have taken the executor count as 7 and the executor memory being 6GB in our experiments.

5.4 Evaluation of Load Balancing Approaches

In this section, we evaluate four different ER approaches. They are Basic approach (discussed in Section 2.2), Block Split, Block Slicer and SBPBS. We will evaluate the performance (Section 5.4.1), speedup (Section 5.4.2), scalability (Section 5.4.3), robustness (Section 5.4.4) and degree of parallelism (Section 5.4.5) of these four approaches by performing different experiments.

5.4.1 Experiment 1: Execution time

In this experiment, we evaluate the performance of all four ER approaches mentioned above. The experiment is performed on a spark cluster with the specifications given below:
In this experiment, we keep the number of executors constant and used different datasets to evaluate the performance for all the four ER approaches Basic, Block Split, SBPBS with 25% sampling. The number of executors used in this experiment is 7, driver memory is 1GB, driver core is 1, executor memory is 6GB, number of partitions are taken as 28, 28, 56 and 56 for SBPBS, Block Slicer, Block Split, and Basic approach respectively based on the results obtained in Section 5.3. The datasets used are DS1, DS2, DS3, DS4, DS5, and DS6 (as given in Table 5.1). Figure 5.5 shows the processing time for all four ER approaches for various datasets.

From Figure 5.5, we can understand that the processing time of SBPBS is less than Block Slicer, Block Split, and Basic Approach. The time taken to process DS1 by Basic Approach is 51% more than the time taken by SBPBS and it is 30% for Block Split. Block Slicer performed better that both Block Split and Basic approach but it still took 22% more time than SBPBS. The highest time difference between SBPBS and Basic Approach is almost 60% which is obtained while processing DS4. The processing time of Block Slicer is up to 30% better than Basic Approach and around 20% than Block Split. Even though Block Slicer performance is similar to Block Split for smaller dataset it performs better for the larger dataset. For dataset DS6 SBPBS is almost 15% better than Block Slicer. The processing time increases.

**Figure 5.4: Processing time for different number of partitions - Block Slicer**

- driver-cores: 1
- driver-memory: 1GB
- executor-cores: 4
- executor-memory: 6GB
- num-executors: 7
5. Evaluation

Figure 5.5: Processing Time for Performance experiment
5.4. Evaluation of Load Balancing Approaches

linearly w.r.t the size of the dataset for SBPBS and Block Slicer, whereas Block Split and Basic approach are not linear and unstable.

5.4.2 Experiment 2: Speedup

In this experiment, we evaluate the speedup for all four ER approaches. Here, we keep the dataset constant and vary the number of executors and calculate the processing time. We have taken the dataset DS2 (given in Table 5.1) for this experiment and calculated the processing time with 1, 2, 3, 4, 5, 6 and 7 executors for all of the three ER approaches. The executor and driver configurations except for the number of executors are same as they are used in performance experiment. Figure 5.6 shows the processing time and speedup for a different number of executors.

![Speedup](image)

From Figure 5.6, we see that SBPBS and Block Slicer has a better processing time both with fewer resources and more resources. With 1 executor Block Slicer performed 8% better than SBPBS and with 2 executors SBPBS performed 7% better than Block Slicer. Overall when compared with Block Slicer, SBPBS performs better than Block Slicer and all three other approaches. Block Split has the highest processing time with both 1 and 2 executors then it picks up its speed once there are
more resources available. This is because of the overhead to calculate the BDM for
the whole dataset when the resources are minimal. Even though BDM calculation
is also an overhead for Block Slicer it only takes more time that basic approach for
1 executor, then it performs better because it’s better load distribution strategy.
Then, both Block Slicer and SBPBS perform better than the basic approach in spite
of the overhead to get block distribution statistics is because the time taken to get
the block distribution statistics is less. Even though there are only 1 or 2 executors
there is a need for load distribution since each executor has four cores. So even
with 1 executor, there will be 4 partitions so we need to distribute the load across
these partitions. The processing time for Block Split reduced by 82.22% when 1
executor is increased to 7 executors. The basic approach had a time reduction of
75%, whereas Block Slicer had 78%. SBPBS has not only better performance but
also a better reduction in processing time almost like Block Split with 81.99%. The
speedup calculated for SBPBS, Block Slicer, Block Split, and basic approach are
5.55x, 4.54x, 5.62x, and 4.07x respectively. This shows both SBPBS and Block Split
has almost same speedup and has around 36% better speedup than basic approach
and 22% better speedup than Block Slicer.

5.4.3 Experiment 3: Scalability

In this experiment, we evaluate the scalability of all the four ER approaches. For
evaluation we used the Dataset DS11 (given in Table 5.1) which a smaller dataset.
We used a smaller dataset because we keep multiplying the dataset with the number
of executors so that the increase in executors will be directly proportional to the size
of the dataset. This helps us to maintain the load vs resource ratio. In order to test
the scalability, we initially start with DS11 datasets for 1 executor and we go until
7 executors where we multiply the dataset with the number of executors i.e., for 2
executors concatenating DS11 twice and so on. We use the same configuration as
used in performance experiment (Section 5.4.1) for executor and driver. Figure 5.7
shows the processing time for the scalability experiment.

From Figure 5.7, it is clear that SBPBS has better scalability out of all four ER
approaches. We can see that Basic approach has a better processing time than
other load balancing approaches. That is because the dataset we chose is small and
has a lesser skew factor, as we explained earlier we chose this dataset so that we
could multiply the dataset. Even though Basic approach has better processing time
for smaller dataset it has the worst scalability out of all four approaches. SBPBS has
the best scalability and Block Split has a better executor vs processing time ratio
but the processing time for Block Split is more than the processing times of SBPBS
and Block Slicer. With 1 executor Basic approach had 14% better processing time
than SBPBS but with 7 executors SBPBS has 40% better processing time than basic
approach.

5.4.4 Experiment 4: Robustness

In this experiment, we test the robustness of the four ER approaches: SBPBS (with
25% sampling), Block Slicer, Block Split and, basic approach. We use the same
executor and driver configuration as it is used in Section 5.4.1. Then, we run all
four ER approaches with larger datasets with higher skew percentage and calculate
5.4. Evaluation of Load Balancing Approaches

Figure 5.7: Processing Time for Scalability experiment
the processing time for each approach. We also find which approach can process
the largest possible dataset (based on the available resources). So we use the same
number of partitions for all the four approaches, driver and executor configuration
used in performance experiment. The datasets used for this experiment are DS5,
DS6, DS7, DS8, DS9, and DS10 (given in Table 5.1). Figure 5.9 shows the graph
and values obtained for the robustness test. Figure 5.8 shows the average block
size and the largest block size in the datasets used in this experiment. The largest
block size for DS5 is 129x more than its average block size, whereas it is 128x, 138x,
143x, 148x and 151x for the datasets DS6, DS7, DS8, DS9, and DS10 respectively.
It also shows the standard deviation of block sizes and the standard deviation of
comparisons in a block.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average Block Size</th>
<th>Largest Block Size</th>
<th>SD(block sizes)</th>
<th>SD(block comparisons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS5</td>
<td>67</td>
<td>8645</td>
<td>415</td>
<td>513218</td>
</tr>
<tr>
<td>DS6</td>
<td>70</td>
<td>8974</td>
<td>439</td>
<td>572418</td>
</tr>
<tr>
<td>DS7</td>
<td>76</td>
<td>10523</td>
<td>486</td>
<td>724751</td>
</tr>
<tr>
<td>DS8</td>
<td>86</td>
<td>12276</td>
<td>556</td>
<td>967093</td>
</tr>
<tr>
<td>DS9</td>
<td>137</td>
<td>20331</td>
<td>914</td>
<td>2637355</td>
</tr>
<tr>
<td>DS10</td>
<td>141</td>
<td>21250</td>
<td>953</td>
<td>2894197</td>
</tr>
</tbody>
</table>

SD -> Standard Deviation

Figure 5.8: Block size information

From the above results, we see that Block Split performs 25% better than basic
approach for DS5 which has $1.4 \times 10^6$ records with 40% duplicate records, whereas
SBPBS performs 14% better than Block Split and 31% better than Basic approach.
Block Slicer performs slightly better than Block Split but SBPBS performs better
than Block Slicer. This shows that Block Slicer does not have a huge performance
boost when the skew factor is not so high. This is because both the approaches
use BDM for storing the block distribution statistics and also processes the whole
dataset. With dataset DS8 containing $2 \times 10^6$ records with 100% duplicate records,
the performance of both Block Slicer and Block Split reduces, and they performs
only 10% and 5% respectively better than Basic approach. But we can see that Block
Split's performance reduced more compared to Block Slicer. This shows that Block
Slicer performs better load balancing than Block Split because its load distribution
strategy which involves less data shuffling between partitions. At this stage, SBPBS
still performs 32% better than Basic approach, 24% better than Block Slicer and 28%
better than Block Split. Both Block Slicer and Block Split failed when tested with
DS9 ($3.3 \times 10^6$ records with 70% duplicate records) and DS10 ($3.5 \times 10^6$ records with
75% duplicate records) whereas the other two approaches completed successfully.
This happens because of insufficient executor memory because of shuffling between
partitions. SBPBS approach keeps performing better and improved its performance
over Basic approach up to 38%. It proves to be a robust approach in spite of
performing data shuffling between partitions to achieve parallelism.
Figure 5.9: Processing Time for Robustness experiment
5.4.5 Experiment 5: Degree of Parallelism

In this experiment, we test how much load distribution these three ER approaches offer. We perform two experiments in order to evaluate this, first we check the size of each partition of the output file that the spark application writes to HDFS and we calculate the time taken by each core of the executors to complete the task. We also calculate the standard deviations of the partitions sizes and processing time for each partition. In order to perform these two experiments we use the dataset DS2 (given in Table 5.1) and set the number of partitions to 28, so that each partition will be assigned to each core once the spark job starts and it helps to find the processing time difference between the core that completes the processing of a partition first and the core that completes last. This information tells us how long the resources are idle. Since we are not worried about the overall processing time we did not use the optimum number of partitions as it would be difficult to precisely compare the time taken by each core and it would not be constant across all four ER approaches.

Evaluation of Partition Size

Figure 5.10 shows the size of each partition of the output file with their standard deviations for all four ER approaches. This gives the information about the amount of data that is processed by each executor core.

![Degree of Parallelism - Partition Size](image)

Figure 5.10: Standard deviation for partition's size.

From Figure 5.10, we understand that the size of the output files partitions is almost equal for SBPBS and Block Split partition size with standard deviations of 6.266 and 6.341 respectively. Whereas Block Slicer’s partition size standard deviation of 101.953 and basic approach has the highest standard deviation of 369.415. In the basic approach, the largest partition is almost 11 times bigger than the smallest partition. This shows that most of the executor cores process too less data while the others process a huge amount of data.

Evaluation of Partition's Processing Time
5.4. Evaluation of Load Balancing Approaches

Figure 5.11 shows the time taken by each core to process one partition. This shows the difference between the first completed partition and the last completed partition. This information helps us to understand how long the resources are idle until the largest partition gets processed. It also shows the standard deviations of processing time calculated for each approach where SBPBS has a standard deviation of 0.367, Block Split has 0.540, Block Slicer has 0.698 and basic approach has 1.842.

![Degree of Parallelism - Processing Time](image)

Figure 5.11: Standard deviation for partition's Processing Time.

From Figure 5.11, we see that the ER approaches those use load balancing i.e., SBPBS, Block Slicer and Block Split has a better resource utilization than the basic approach. For the basic approach, the smallest partition took only 1 minute to process it whereas the largest partition took 8.9 minutes which is 7.9x more than the time taken for the smallest partition. This means the executor core assigned to the smallest partition is idle for 7.9 minutes which is 89% of the overall processing time. Meanwhile, in Block Split the processing time of executor core that took more time and less time is 5.5 minutes and 3.6 minutes respectively. So the executor core with highest processing time is 1.52x of the executor core with least processing time. Whereas for Block Slicer the least processing time of a core is 2.6 minutes and the highest is 5.4 minutes which is 2.07x of least processing time. Even though the highest processing time is slightly less than Block Split. The least processing time is 25% less than Block Split which shows that the degree parallelism for Block Split is better than Block Slicer. In SBPBS, the highest and the least processing times are 5.1 minutes and 3.7 minutes respectively. So the executor core with highest processing time is 1.37x of the executor core with least processing time. This shows that SBPBS has a better load distribution than Block Split and Basic approach. Not only the difference between the least and the highest processing time, but even the standard deviations calculated show that SBPBS with the standard deviations of 0.367 is better than the other three approaches. Even though Block Slicer proved to perform better than Block Split it has slightly a higher standard deviation than Block Split. Basic approach has the highest standard deviations with 1.842 which shows that the load distribution is not properly done.
From both the results, it is obvious that the proper distribution of load on the partitions has a huge impact on the processing time.

5.5 Summary

In this chapter, we evaluated SBPBS, Block Slicer, Block Split and basic approach based on different metrics: performance, speedup, scalability, robustness, and degree of parallelism.
6. Related Work

In this chapter, we discuss about the related work in the field of Entity Resolution (ER), parallel-ER approaches and approaches proposed to handle data skew. In Section 6.1, we discuss about the research work related to the general ER problem. In Section 6.2, we discuss general sampling-based data skew handling strategies. In Section 6.3, we discuss about the existing approaches to handle data skew problem in parallel-ER, which are exactly the existing approaches to solve the proposed problem of this thesis.

6.1 Entity Resolution

Entity Resolution has been one of the strongest research areas for more than forty years [30]. Ever since its first introduction in 1959 for linking birth records [51] a considerable amount of contributions are made to this field. One of the major contributions was made by Winkler which is Expectation-Maximization [52]. ER was done in a naive way where one record was compared with every other record in the dataset which was very inefficient, [53] proposed an approach known as blocking mechanism to reduce the search space.

Parallel Entity Resolution Parallel computing has drawn a lot of interest in the recent years because of its optimal use of resources and reduced processing times. There are several approaches which are proposed for parallel-ER [48, 54, 55] by scaling up the resources. Although these approaches provide parallelism and contribute to the reduction of processing time. It leads to a more severe problem, data skew (discussed in Section 2.5). Parallel ER is implemented using two distributed computing frameworks namely Apache Spark and Hadoop MapReduce [12, 56]. In [57], Apache Spark is used to implement entity resolution application for Adverse Drug Reaction (ADR) databases which is maintained by World Health Organization (WHO) and drug regulators in many countries [57]. They used Spark as a distributed framework and k-Nearest Neighbor (kNN) to classify the records.
6.2 Sampling-Based Data Skew Handling Strategies

In this chapter, we discuss the strategies based on sampling to handle the data skew problem. The current state-of-the-art approaches namely Block Slicer and Split need to process the whole dataset to get the statistics of key distribution, this creates an overhead to the ER process. In order to overcome this Yujie Xu [58] proposed two partitioning schemes based on sampling the dataset to collect distribution statistics. They are cluster combination optimization and cluster partition combination for handling less and highly skewed datasets respectively. This approach uses a parallel sampling approach to sample the input dataset and perform the actual task beforehand in order to generate expected intermediate keys and their sizes. A similar and advanced algorithm called CORP was proposed by Zhuo Tang [59]. This approach uses reservoir sampling for sampling the input dataset and generates a distribution matrix based on the intermediate results. This matrix will then be used to schedule map and reduce tasks on the nearby nodes.

A Spark based general load balancing algorithm was proposed in [43], this algorithm uses Reservoir Sampling to collect key distribution information and use splitting and combining algorithm to achieve load balancing. The skewed keys are predicted based on an approximation method and they are split using the splitting and combining algorithm.

6.3 Strategies for parallel Block-Skewed Entity Resolution

In this section, we discuss the parallel-ER approaches in MapReduce and Spark. These approaches are introduced to distribute the load evenly across all the nodes in the cluster. In Section 6.3.1, we discuss MapReduce based ER load balancing techniques. In Section 6.3.2, we discuss the general load balancing techniques in Spark and load balancing techniques for ER.

6.3.1 MapReduce-Based ER Techniques

Since its release in 2008 MapReduce have gained huge popularity in the field of parallel computing. It seemed to be a perfect solution for the ER as the amount of data processed in the ER process increased exponentially. Many researchers in the field of ER started experimenting with MapReduce. They realized that the most important thing to be considered while using MapReduce framework for ER is proper distribution of load. Lars Kolb [12] explained the problem of skewed data and proposed two load balancing approaches BlockSplit and PairRange to overcome data skew problem. But the limitation of these approaches is that they are not scalable. So Wei Yan [60] proposed two scalable solutions Cell Block and Cell Range which are similar to Block Split and Pair Range. Improved load-balancing algorithms, BlockSlicer [18] and MSBlockSlicer [61] are proposed to have better load-balancing. We discuss Block Split and Block Slicer in detail below:

Block Split [12] and Block Slicer [11] are two popular blocking based load balancing techniques for ER.
Block Split distributes the load by creating one or more match tasks (a group of records to be compared) per each block and distribute these match tasks across the reducers. It splits the larger blocks and distributes the sub-block which will be considered as an individual block. This splitting is done by generating a composite key which includes information about the partition on which the records are stored. Once the composite key is assigned the blocks are reduced by their keys. This makes sure the records are distributed uniformly. Block Slicer uses a similar approach but it does not depend on the partition information [18]. It groups the records and splits the larger blocks using a similar technique as Precise Block Slicing explained in Section 3.3, but it does not split blocks precisely and the sub-blocks are of uneven size. In Block Slicer approach the amount of replication is less compared to Block Split.

As we discussed in Section 3.2, we need to obtain block distribution statistics in order to split and distribute the blocks. Both of the state-of-the-art approaches Block Split and Block Slicer use a data structure called Block Distribution Matrix (BDM) to store the information about the number of records in each block and, also which partition they are stored and on which node they are located. This will be a $b \times p$ matrix, where $b$ is the number blocks and $p$ is the number of partitions. The size of the BDM increases as the number of blocks in the dataset increases. This can be seen as a limitation as the this BDM is stored in-memory and if the size of BDM increases it cannot scale when the dataset is huge.

### 6.3.2 Spark-Based Load Balancing and ER Load Balancing Techniques

In master thesis work [56], Block Split [12] and Pair Range [12] load-balancing algorithms are implemented using Apache Spark. There is not much research work w.r.t entity resolution using Spark. In this thesis work, we focus on load distribution for entity resolution using the Apache Spark framework and also using a sampling approach to reduce the pre-processing time to collect key distribution statistics.
7. Conclusion and Future Work

7.1 Conclusion

In this thesis work, we discussed Entity Resolution(ER), data skew problem in ER in a distributed environment and proposed an ER load balancing algorithm called \textit{Sampling-Based Precise Block Slicer (SBPBS)} to handle data skew problem. We evaluated SBPBS against other ER load balancing approaches namely Block Split and Block Slicer, and also against the basic ER approach without load balancing. Both Block Slicer and Block Split use a special data structure called Block Distribution Matrix(BDM) to store block distribution statistics whereas SBPBS uses stratified sampling to collect samples from the input dataset, and calculates each block size and stores it in a HashMap as key-value pairs. The block distribution statistics collected is used to calculate a threshold value based which larger blocks (w.r.t number of comparison) are split into smaller sub-blocks.

For evaluating these ER approaches, we performed various tests to evaluate the performance, speedup, scalability, robustness, and degree of parallelism. We used the datasets with input records of order $10^6$ with a varying skew percentage and one $10^5$ order dataset which is used for scalability experiment.

The results obtained for performance evaluation experiment shows that Block Slicer is 30\% better than Basic approach and 20\% better than Block Split. But SBPBS proves to be 15\% better than Block Slicer. Thus, it is clear that SBPBS performs better than the other three ER approaches. The results we obtained show that the basic approach had 4.07x speedup, Block Slicer had a speedup of 4.54x. SBPBS and Block Split had the highest speedup among all four ER approaches with 5.55x and 5.62x respectively. In the scalability experiment, we saw that all there load balancing approaches had better scalability than the basic approach which shows the importance of load balancing. Out of the three ER load balancing approaches SBPBS had better scalability.

We also evaluated the robustness and degree of parallelism. For robustness, we used 7 executors and kept increasing the size of the dataset. Both Block Split and Block
7.2 Future Work

Slicer couldn’t process DS9 and DS10. Whereas basic approach and SBPBS could process both of these datasets. For DS10, SBPBS had 38% better performance than basic approach. For evaluating the degree of parallelism we calculated the difference between the processing time of the partition that completed first and the partition that completed last. The difference is 7.9x for Basic approach, 2.07x for Block Slicer, 1.52x for Block Split and 1.37x for SBPBS. We also calculated the variance of partition's processing time for all four ER approaches and the variances obtained for SBPBS, Block Slicer, Block Split, and basic approach are 0.367, 0.698, 0.540 and, 1.842 respectively.

7.2 Future Work

In the given limited time for the thesis work, we proposed and implemented a load balancing approach for ER and evaluated our approach against other state-of-the-art approaches. We discuss the possible future work that could be done to further evaluate our approach and work that could be done to improve load distribution in a distributed environment below:

**Adding Up Smaller Blocks:** Our approach SBPBS splits the larger blocks but leaves the smaller blocks as it is. Even though we distribute the blocks as even as possible using "partition pool" (discussed in Section 4.1.2). There will be a slight difference in the number of record-pair comparisons performed on each partition. So, we could make a few changes to SBPBS to add all the smaller blocks together to create a block with comparisons that will be approximately equal to the average number comparisons and evaluate it with SBPBS approach.

**Different Type of Evaluation for Robustness:** In our current robustness evaluation, the datasets we used had an increased number of duplicates records which eventually increased skew factor in the dataset. We could also test our approach by generating datasets a precise skew factor from 0% to 100% where the percentage of the skew factor defines the size of the largest block in the dataset.

**Evaluation against Other ER Load Balancing Approaches:** In our work, we evaluated SBPBS against other ER load balancing approaches those use BDM. We could also evaluate SBPBS against Cell Block ER load balancing approach that uses an approximation strategy to store block distribution statistics.

**Improving Spark Load Balancing:** Even though Apache Spark can perform faster because of its in-memory computation it lacks a good load balancing strategy when it comes to handling skewed data. Thus, contributing to Apache Spark’s load distribution strategy will help to eliminate the need for implementing separate load balancing algorithms for various applications to a great extent. This kind of approach is proposed in [62].
Bibliography


Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den February 13, 2019