Load Balancing Strategies for Spark-Based Entity Resolution

Author:
Kirity Rapuru

May 24, 2018

Advisors:
Dr.-Ing. Eike Schallehn
Institute of Technical and Business Information Systems

M.Sc. Xiao Chen
Institute of Technical and Business Information Systems
Rapuru, Kirity:
Load Balancing Strategies for Spark-Based Entity Resolution
Abstract

Distributed computing is opening up many avenues in the age of big-data, as it is capable of processing work-loads otherwise cannot be contained on an individual machine. Various programming models are introduced to support distributed computing paradigm. Among them, Hadoop’s MapReduce [1] and Apache Spark [2] are the two most adopted frameworks within the big-data community [3]. Entity Resolution (ER) is the task of detecting digital-records (objects), which refer to the same real-world entity. Due to its innate pairwise comparative nature, inevitability ER is a pervasive, resource-intensive, data-intensive process, even on modern high-end hardware. Hence, ER with distributed processing is gaining a lot of attention. Although ER can leverage distributed computing, the efficiency and scalability depend on, how uniformly the load is distributed among the worker nodes. In order to fully utilize the novel approach of distributed computing, uniform distribution of load, even for the skewed data is of paramount importance. In this work, we analyze the load-imbalance of ER in Spark’s environment set up. To improve execution times w.r.t to skewed data, we have adopted two load-balancing strategies for Spark’s environment, which work in tandem with blocking techniques. The results show that our adaptations increased the performance of the overall process.
Acknowledgements

First, I would like to thank Dr-Ing. Eike Schallehn for providing me the opportunity to write this thesis under his supervision. I extend my deepest gratitude to my advisor Xiao Chen, who has given me this topic of my interest too. Her constant guidance and valuable discussions are only the main cause for the progress of this topic. I thank once again from my heart for her patience and availability during the whole duration of this thesis. Second, I would to thank Prof. Dr. rer. nat. Frank Ortmeier for giving me the admission to study in Digital Engineering at OVGU and whole OVGU, in general, who supported my studies in Germany.

I would like to thank M.Sc. Gabriel Campero Durand for his support in this work.

Further, I would like to thank, my brother Sudarsan Prathipati who supported me in all regards to the completion of my Master’s, my sisters: Keerthana Rapuru, Subhashini Prathipati, and Nirmala Thotakuri, and of course the whole-hearted support of my parents without whose contribution this may not be possible. I would like to thank all my friends and fraternity, who were part of my studies directly or indirectly. I would like to special mention: Herr. Kowol Siegfried, Prabhu Appalapuri, Hari Kumar Guduri, Nishanth EV and veeranjaneyulu punati. Finally, I would like to offer my thanks and gratitude to all the people who helped me 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, May 23th 2018

Kirity Rapuru
5 Evaluation
  5.1 Experimental Setup ............................................. 53
    5.1.1 Datasets .................................................. 53
    5.1.2 Cluster environment ........................................ 55
    5.1.3 Submission of Spark Application to the Cluster .......... 56
  5.2 Experiment I: Finding optimal partitions and tuning the parameters . 57
    5.2.1 Tuning of Basic Approach ................................... 57
    5.2.2 Tuning of BlockSplit Approach .............................. 57
    5.2.3 Tuning of PairRange Approach .............................. 59
  5.3 Experiment II: Robustness ....................................... 61
    5.3.1 Type I .................................................... 62
    5.3.2 Type II .................................................... 63
  5.4 Experiment III: Speed-up and Scalability ...................... 64

6 Conclusion and Future Work ........................................ 67
  6.1 Conclusion ..................................................... 67
  6.2 Future Work .................................................... 67

Bibliography .......................................................... 69
List of Figures

2.1 Extracted Charlie Chaplin’s information from Wikipedia and Britannica. .................................................. 7
2.2 Online price comparison of mobile phone from a website Smartprix [5]. 8
2.3 Soundex character to numeric conversion table. ......................... 11
2.4 Fuzzy Soundex character to numeric conversion table. .................. 12
2.5 Classification block diagram of record matching process [6]. ............ 18
2.6 Apache Samza architecture (left) along with Apache Hadoop architecture (right) [7]. ................................... 21
2.7 Spark’s components stack. .................................................. 22
2.8 Spark’s architecture [8]. .................................................... 25
2.9 The three ways of Spark's execution [9]. ................................. 26
2.10 Anatomy of Spark’s jobs execution [10]. .................................. 27
4.1 Flowchart of baseline approach of ER, as described in Section 4.1. . . . 33
4.2 Blocking keys of the running example with five records. ................. 34
4.3 Comparison step with similarity values. ................................. 35
4.4 Sample output of classification step of running example. ............... 35
4.5 Task execution times of the Basic approach with DS3. ................. 36
4.6 Block Distribution Matrix computations steps, adapted from [4]. . . . 38
4.7 Block Distribution Matrix. ................................................ 39
4.8 Flowchart of the BlockSplit approach. ................................. 42
4.9 The BlockSplit approach work flow. .................................... 43
4.10 Enumeration of all pairs of running example, adapted from [4]. Three ranges are indicated with three different color coding. ........... 45
4.11 Flowchart of the PairRange approach. ................................. 48
4.12 Flow of PairRange working example. .................................. 50
5.1 Test datasets used for evaluation. .................................. 54
5.2 Execution times and partitions for the Basic approach. .......... 58
5.3 The jobs and stages information of the three strategies. ........... 58
5.4 The effect of match_tasks ($m_t$) on the execution times of the Block-Split approach. ........................................... 59
5.5 Task times for various match_tasks of BlockSplit approach. ...... 60
5.6 Execution times and ranges ParRange approach. .................. 60
5.7 Task times and ranges of ParRange approach. .................... 61
5.8 Execution times at different data skew percentages with DS2. ...... 62
5.9 Execution times and different data skews with DS2. ............... 63
5.10 Execution times and speedup values for all the three approaches using dataset DS4. ................................................. 64
5.11 Tuning of DS4 for BlockSplit and ParRange. .................... 65
List of Tables

2.1 Example census data. .................................................. 6
2.2 Suffix array based example with four names, modified from [64]. ... 14
2.3 Suffix array based example with four names, modified from [64]. ... 14
2.4 Features comparison of Sparks API. ................................ 23

4.1 Example dataset with three rows. ................................. 32
4.2 Execution and configuration details of the Basic approach. ... 36
4.3 Dataset with partitions and with blocking key. ................. 37

5.1 Cluster nodes configuration and version information. ........... 55
1. Introduction

Distributed computing is receiving a lot of attention and becoming a popular paradigm in the recent past for the adaptation of resource and time-intensive workloads. Especially, in the era of big data [11]. It is seen as the solution for the processing, analyzing and getting insights into to magnanimous amounts of data being generated. In the past, data are generated or captured for only finite manual interventions, but nowadays data are also generated by machines (IOT devices) and collected in a continuous fashion too. This shift has led to reaching the limits of traditional and sequential processing of data. In this background, distributed computing has become important to perform the large work-loads in parallel (otherwise may not be practical for some use cases) and also to gain increased speeds.

Programming model like MapReduce [1] is been introduced to facilitate distributed computing paradigm in 2008, in a largely transparent manner [4]. MapReduce along with HDFS (storage layer) provided an effective programming model for the parallel processing in the distributed environments. The combinations of MapReduce, HDFS and supporting packages are collectively called as Hadoop [12] ecosystem. Hadoop has become the de facto tool for big data processing and analytics, as it provided simple APIs, hiding complexities behind fault-tolerance and parallel executions. In MapReduce, all the intermediate results are saved to disk, due to this, multi-stage processing systems and iterative work-loads were taking additional times for reading and writing to disks, apart from actual work.

To overcome this limitation of repeated reads from the disk with MapReduce, a paper [2] on Apache Spark was published and a stable version software [13] is released in 2013. Spark introduced Resilient Distributed Datasets (RDD), which are an in-memory (main-memory) data structures and are capable of caching intermediate results without writing to the disk. This new feature has increased the speed of multi-stage programs and iterative type work-loads considerably.

Entity Resolution (ER) is the task of identifying digital-records, which refer to the same-real world entities [6]. In other words, it is the process of determining the duplicates (near duplicates) in a given dataset. Therefore, it has many real-world applications, which includes data integration, enhancing data quality, bio-informatics,
shopping-cart comparisons, threat analysis, linkage of census data, plagiarism etc. In the literature [14–17], ER is also referred as entity matching, record linkage, duplicate detection or reference reconciliation.

1.1 Motivation

In this thesis, we work with the challenges relating to ER, which are data-intensive and performance critical in nature. To understand the massiveness of ER, a few problem statements are considered, which are to be solved in day-to-day applications in the real world: the task of linking of census data of a country, threat analysis, matching of a person’s DNA with all the suspicious person’s databases around the world. These real-world examples, fall into the bracket of big data, with the order of records ranging millions to billions. Hence, ER is an important task, which needs attention w.r.t to improving matching quality and quickness of the results.

The challenge regarding ER is ubiquitous, as every record in the dataset needs to be compared (matched) with every other record within the dataset. In naive approach, the matching technique is to be applied on the Cartesian product of all the input records leading to $O(n^2)$ complexity. In case of large datasets, such an approach is not very efficient. In order to decrease the search space caused by the Cartesian product, blocking techniques (detailed explanation in Section 2.2.2) are dominantly used [18]. However, even after the application of blocking techniques, execution time of ER can range from days - weeks [19].

The process of blocking not only reduces the search space but also opens up the possibility of blocking independently (if feasible), as the idea of blocking is to just compare the records within the blocks. Therefore, ER is an ideal use-case for the distributed computing models. After blocking, the other possibilities to improve the execution times include: improving the join strategies within the blocks, improving the comparison techniques, improving the load distribution across the nodes in the cluster, and improving blocking technique itself. In this thesis, we work on the improvement of load-balancing aspect in the distributed environment.

Normally, all the distributed computing models has a default data distribution algorithms inbuilt in them. However, ER has an inherent challenge of data skew (detailed explanation in Section 2.3) in many real-world datasets. Due to this, blocks generated after the blocking step will not have uniform sizes. As the computations of blocks are spread across the nodes, with the default distribution provided by distributed computing models, few blocks (in general) will dominate the response time of the system. Large blocks will become the bottlenecks on some nodes. As the skew factor increases, the problem aggravates leading to inefficient usage of resources in the cluster. In our work, we study the data skew problem on ER and Spark as the distributed computing environment, to adopt load balancing strategies to improve the efficiency.

1.2 Goal of the Thesis

The goal of the thesis work is to remove performance bottlenecks and improve the execution times, thereby, improving the efficiency too. To achieve our goal, we have breakdown our total work into the following logical steps:
• Analyzing Sparks execution plans intricately and figuring out the most dominating reasons for the higher execution time for skewed datasets.

• Select possible load balancing strategies to address the challenge of higher execution times and adopting them for the Spark environment.

• Finally, evaluating all the strategies against performance, robustness and speed-up with synthetically generated datasets and comparing with the baseline approach—which is not tailored for load-balancing problem.

1.3 Structure of the Thesis

This work is divided into six chapters and organized as follows:

**Chapter 2.** In Chapter 2, we introduce the topics necessary to understand this work. Therefore, we will explain ER and it’s applications, steps involved in ER, data skew and parallel computations engines. The aim of this chapter is to sufficiently provide background knowledge of ER and Spark as parallel computation engines, which are key parts of this work.

**Chapter 3.** In Chapter 3, we provide comprehensive research work related to our work. Here, various approaches, standpoints from various papers, journals and books by numerous authors are presented. The aim of this chapter is present the latest up to date literature related to our work.

**Chapter 4.** In Chapter 4, we describe the challenges facing by ER in our selected parallel computing framework, next we introduce two approaches to improve the aforesaid situation with a running example. The aim of this chapter is to propose and explain the problem definition and our approaches in detail.

**Chapter 5.** In Chapter 5, we evaluate our approaches with three important metrics: robustness, performance, and scalability. The aim of this chapter is to evaluate and discuss the effects of the proposed approaches on ER.

**Chapter 6.** In Chapter 6, we summarize our results, findings and provide an outlook for the future work.
1. Introduction
2. Background

In this chapter, we introduce the concepts and work-flow of Entity Resolution (Section 2.1), the problem of data skew (Section 2.3), and various distributed computing engines (Section 2.4), which are required to understand this thesis.

2.1 Entity Resolution

In this section, we cover the definition of Entity Resolution (Section 2.1.1), applications of Entity Resolution (Section 2.1.2), and the process of Entity Resolution (Section 2.2) in detail.

2.1.1 What is Entity Resolution?

Entity Resolution (ER) is the process of identification of identical entities of the same real-world objects in a given data source(s). ER is also popularly known as entity matching, record linkage, de-duplication, data matching, object identification [14], reference reconciliation [15], merge/purge [16], identity uncertainty [17] etc. In general, ER is used for enhancing the data integrity and quality.

2.1.2 Applications of Entity Resolution

In order to project the importance of ER and current usage in real-world applications. Next, we present five major and diverse regions where ER is applied: data integration, linking of census data, lining hospital/patient data, online shopping comparison respectively.

- **Data Integration:** The aim of data integration step is to merge or combine or integrate multiple data sources into a single data source [15]. The familiar obstacle in data integration process arises when entities from disparate sources describe the same real-world entity. Then, duplicate entities from disparate sources should be recognized and merged into a single object. The job of detecting duplicates is done by the ER process.
2. Background

<table>
<thead>
<tr>
<th>Name</th>
<th>DOB</th>
<th>State</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith, William</td>
<td>1/2/73</td>
<td>Main street</td>
<td>Berkeley, California</td>
</tr>
<tr>
<td>Smith J, William</td>
<td>1973.1.2</td>
<td>Main St.</td>
<td>Berkeley, CA</td>
</tr>
<tr>
<td>Smith James William</td>
<td>Jan 2, 1973</td>
<td>16 Main street</td>
<td>Berkeley, Calif.</td>
</tr>
</tbody>
</table>

Table 2.1: Example census data.

When the data are collected from different data sources like Wikipidea\(^1\) and Encyclopædia Britannica\(^2\), same person or entity could be represented in different ways. For instance, a renowned British actor *Charlie Chaplin* is shown in Figure 2.1, which is from both the sources. In Wikipedia, there are some additional details of *occupation*, *years active* and *spouse(s)* details. Hence, merging of two sources would result in more detailed and unified view of Charlie Chaplin.

- **Linking of census data:** The common challenge faced by statistical institutes and census collection organizations is, integration of census data sets collected from disparate sources [20]. It also becomes challenging due to the following reasons [21]:
  
  - Due to change of residence address of an individual over the period of time.
  - Changes in names of streets or cities over a period of time.
  - Re-organization of administrative provinces or regions.

- **Linking Hospital/Patient Data:** The fatality and ephemeral nature of health emergency-care mean, a single database is not going to sufficient for the injury based research studies, based on the population. In case of small applications, an accurate human judgment can be made to decide, whether two records belong to the same patient or not. However, practically it is not a feasible solution to depend on the human judgment of matching in case of large database situations. Hence, naturally, there is an inevitable need for software-based linking solutions for matching purposes [22]. Health service data are normally recorded from the stand-point of administrative purposes. But, this data can also serve the purpose of research in epidemiology, monitoring services and evaluation of health. Linked data can be used in health surveillance systems, suspicious pattern detections, an outbreak of epidemics, enrich the healthcare for patients [23, 24].

- **Online Shopping Comparison:** In the recent past, there is a significant increase in online shopping (eCommerce) culture. Due to this, many vendors across the spectrum, who may use different text descriptions to describe the same product (also referred to as “offers”). Thus, on-line comparison (maybe prices) engines face the challenge of identifying the same product by various

\(^1\)https://www.wikipedia.org/
\(^2\)https://www.britannica.com/
### Charlie Chaplin
BRITISH ACTOR, DIRECTOR, WRITER, AND COMPOSER

**Sir Charles Spencer Chaplin**

**Publicity portrait, c. 1920**

**Born**
Charley Spencer Chaplin
15 April 1889
England

**Died**
25 December 1977 (aged 88)
Corsier-sur-Vevey, Vaud, Switzerland

**Occupation**
Actor, director, composer, screenwriter, producer, editor

**Years active**
1899–1976

**Spouse(s)**
- Mildred Harris (m. 1918; d. 1920)
- Lita Grey (m. 1924; d. 1927)
- Paulette Goddard (m. 1936; d. 1942)

---

**From Wikipedia**

**Also Known As**
Sir Charles Spencer Chaplin

**Born**
April 16, 1889
London, England

**Died**
December 25, 1977 (aged 88)
Corsier-sur-Vevey, Switzerland

---

**From Encyclopedia Britannica**

---

Figure 2.1: Extracted *Charlie Chaplin's* information from Wikipedia and Britannica.
vendors. This problem in this domain is also referred as *product normalization*. Only after solving this problem, the comparison engines are capable to show the multiple offers for a product to their customers. This comparison eventually helps the customers decision regarding the selection of vendor. For accurate calculation of pricing trends in market place, accurate product normalization is an important task [25].

Figure 2.2 illustrates the price comparison of mobile phone “iPhone 8” from a price comparison website³. This on-line comparison website is comparing the product of interest from five different vendors: namely Flipkart [26], Amazon [27], Tata CLiQ [28], eBay [29], and ShopClues [30]. The comparison is performed on critical factors like “Sellers Rating”, “Shipping Time” and most importantly on “Price”.

<table>
<thead>
<tr>
<th>Seller</th>
<th>Seller Rating</th>
<th>Shipping</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flipkart</strong></td>
<td>★★★★★</td>
<td>Shipping: FREE 2-5 Business Days</td>
<td>Rs. 55,999</td>
</tr>
<tr>
<td><strong>amazon.in</strong></td>
<td>★★★★★</td>
<td>Shipping: FREE 4-6 Business Days</td>
<td>Rs. 54,990</td>
</tr>
<tr>
<td><strong>Tata CLiQ</strong></td>
<td>★★★★☆☆</td>
<td>Shipping: FREE 2 Business Days</td>
<td>Rs. 58,990</td>
</tr>
<tr>
<td><strong>eBay.in</strong></td>
<td>★★★★☆☆</td>
<td>Shipping: FREE 4-7 Business Days</td>
<td>Rs. 57,890</td>
</tr>
<tr>
<td><strong>ShopClues</strong></td>
<td>★★★★☆☆</td>
<td>Shipping: FREE 4-6 Business Days</td>
<td>Rs. 60,300</td>
</tr>
</tbody>
</table>

Apple iPhone 8 Price in India is Rs. 54,990. You Can Buy Apple iPhone 8 Online After Finding Lowest Price Here.

![Figure 2.2: Online price comparison of mobile phone from a website Smartprix [5].](image)

**National Security Analysis:** Post terrorism attacks on the United States of America (USA) in September 2001, there were massive efforts and financial allocations into sophisticated analytics programs. The aim was to identify such threats in beforehand and prevent them in future [6]. Terrorists are much harder to track, in contrast with traditional armies, because of the loose organization of their hierarchy (deliberately) and receiving financial aid from a variety of sources [31]. However, terrorists do leave traces regarding their communications and fundings, in a complex web of Internet space.

The primary aim of counter-terrorism bills, like *total information awareness* (TIA) and the *multistate anti-terrorism information exchange system* (MATRIIX) [32] was to detect suspicious patterns of activity from the records.

³https://www.smartprix.com/mobiles/apple-iphone-8-p1101r03th5u; visited on March 10th, 2018
(transactions) from public and private institutions like airline services, mobile service, financial services, flight schools, and banks.

The challenges of applications of advanced analytic techniques in the areas of record linkage, image processing, natural language processing, and in biometrics are manifold, due to following reasons:

- Typically records to be processed are in the order of millions.
- The data should be searched across disparate heterogeneous data-sources.
- In the timeline of data collection, naturally, there is a possibility of changes in attributes like address, marital status, working labels etc.
- The data are collected across different geographical locations.
- People with criminal intentions and terrorists, are most likely use false information or morphed identities [33].

By 2008, after active application of the above counter-terrorism measures, USA was maintaining the suspicious persons’ list close 500,000 [34].

- **Others:** ER is widely used in many other diverse applications, apart from above-mentioned use cases. Few among other are, improving data quality by database de-duplication, data matching across organizations, linking of taxation records with social welfare schemes, fraud and crime detection [35]. Linking of consumer products, bibliographic citations [36], web searches, genome sequences, publishing of linked data [21, 37], mirror detection [38, 39], anaphora resolution [40].

## 2.2 Entity Resolution Process

ER process consists of five steps and depending on the requirement and application, some these steps can be curtailed. The steps involved are data pre-processing, blocking, field and record comparison, classification and evaluation. Next, in this section, all the five steps are described in detail.

### 2.2.1 Data Pre-Processing:

In general, most real-world data consists of inconsistent, inaccurate, incomplete, noisy, and missing data [41–43]. According to a study, it is estimated that up to 12% of institutions revenues are lost due to data quality related problems [44]. For any data analysis and management system, *garbage-in garbage-out* proposition is applicable. This means, the quality of output data depends of the quality of input data. There are many dimensions to describe data quality of any database. These dimensions are accuracy, completeness, timeliness, accessibility, believability [6]. The most relevant causes for the data quality problem w.r.t to entity resolution are shortly described below:

- **Multiple input sources:** In case, data are recorded by various institutions, different systems, with different data recording modes [45], at different locations, at different time-lines, then probability of data inconsistency are more.
2. Background

- **Limited computational capacity:** As data matching is an expensive task, if the matching sets of database(s) are large and if computational and storage capacities are spare then advanced data matching algorithms are not viable to run. In such cases, basic matching algorithms must be employed, which might not produce high quality results. And this low quality and not accurate enough output, may not be suitable for certain type of applications, which demand high quality. The adaptation of cloud based computing and storage also may not be a practical option in every case, due to security and privacy concerns.

- **Coded representation:** Data could stored in *coded* formats in databases across different domains or different disciplines or different organizations. If data matching is performed between these databases, then careful decoding (mapping) of formats, abbreviations are required before actual data matching.

- **Complex representations:** Many conventional entity resolution techniques are designed over character based formats (name, address) or numeric based (date, age). However, increasingly data are stored in more complex formats, like XML [46], linked entities. Data matching techniques should also consider handling of such complex representations.

- **Restrictive rules:** This condition can result in *inaccurate and low quality* data in database systems. For instance, in case of emergency department registration form has *date of birth or age* as compulsory attributes. Then, in case of unconscious or semi-conscious patients, there is a high probability of entering age attribute as a guess work and defaulting the ‘date’ and ‘month’ to ‘01’. In these kind of situations, there will be an unusual number of patients, whose date of birth is ‘01-January’.

2.2.2 Blocking

The process of blocking involves, generation of a blocking-key for every record in the dataset. Then, records with the same blocking-key are compared. Next, we introduce various techniques of generating blocking-keys (in Section 2.2.2) and various blocking techniques (Section 2.2.2).

**Blocking-Key Values (BKV) Generation Techniques**

An experimental survey [64] emphasizes that construction of blocking key is more important than the type of blocking approach implemented [35], hence choosing of blocking-key generation technique becomes important. Blocking-key is an additional attribute generated for each row in a database(s). Since, most of the real-world data contains errors (of varying percentages), blocking-keys are generated, by considering different set of attributes. Thereby, the chances of potential identical records being slipped away will be mitigated. With this discussion, various ways of blocking-key generation techniques are discuss below.

Blocking-key is generated by applying a phonetic encoding function to a column in the database. The selection of a particular column (or columns) is done by a
domain expert. The idea of applying phonetic encoding functions is based on the proposition that string attributes will be converted into a numeric code, thereby errors (and noise) will be compensated (how this is achieved is discussed below). Three phonetic based encoding techniques are discussed below:

1. **Soundex**: Simple and computationally efficient Soundex [47, 48] is one of the most adapted sound based encoding algorithm. It is developed based on American English language. This algorithm converts all the characters of a string into corresponding numbers (according to Figure 2.3) except first character.

   - All zero values (i.e. ‘h’, ‘w’ and ‘y’) are removed.
   - Repetitions of numbers are removed.

2. **Double Metaphone**: Phonetic based BKV generation algorithms like Soundex and other ([49], [50]) are designed, specifically for English language names. This is a significant shortcoming, if the databases of interest contains names non-English names. Double Metaphone algorithm is designed by considering European and Asian names [51]. The major differences when compared to phonetic based algorithms are: (a) It generates a stream of characters than numbers (unlike phonetic based), (b) For certain names two phonetic transformations are generated, unlike always one. For instance, with Double Metaphone, the name ‘kuczewski’ is transformed to ‘kssk’ and ‘kxfsk’.

3. **Fuzzy Soundex**: This technique employs q-gram related pre-processing with Soundex conversion table [47]. Q-gram are generated by taking sub-strings from the string attributes. If value of ‘q’ is two, then they are called as bigrams and if ‘q’ is three, then called as trigrams. In this method, substitutions are done with q-grams (as substitutions done with numerics in above algorithms). Full conversion table of values are shown in Figure 2.4. Q-gram based approach achieves better quality results compared with Soundex techniques [52].

### Blocking Techniques

Even in the case of small databases, the majority of record comparisons pertains to non-matching records [6]. To improve this situation, where the majority of the matches will come into the category of matching, **blocking** will be done before actual comparison. The objective of blocking step is to decrease the search space of the
subsequent matching process to a feasible extent. This could be achieved by eliminating the unalike (dissimilar) comparisons in actual comparison process. Next, we introduce five blocking techniques:

1. **Standard Blocking Algorithm**: This algorithm is been used for decades in the area of data matching [53]. In this algorithm, each record is grouped into a single block only. In this technique, for each record in the database a BKV is generated and records with same BKV is grouped into same block. To improve the effectiveness, an inverted index structure [54, 55], with BKV as key is created. For simple understanding of the number of candidate pairs generated with this indexing approach, let us consider the following example [35]. Let \( m \) and \( n \) be records in two databases, respectively. Number of BKVs common is \( b \). Then, each block consists of \( m/b \) and \( n/b \) records, respectively. Let \( c \) be the candidate pairs generated, then \( c \) is

\[
c = b \left( \frac{m \cdot n}{b \cdot b} \right) = \frac{m \cdot n}{b} \tag{2.1}
\]

For a single database with \( n \) records, \( b \) BKVs, \( c \), is

\[
c = b \left( \frac{n \cdot (n-1)}{b \cdot b} \right) / 2 = \frac{n \cdot (n-1)}{2b} \tag{2.2}
\]

2. **Sorted Neighbourhood (SNH) Approach**: Alternative techniques to standard blocking was proposed by Hernandez and Stolfo [16, 56]. SNH approach generates ‘sorting key’ (similar to blocking-key). It sorts database based on sorting key, rather than generation of blocks of BKVs. Then a sliding window is defined of size \( w \), then window is slided over the sorted database. Then, the candidate pairs are produced from the records, which are in the sliding window in any given step.

In case, if two databases are used in matching, then first both are merged in first step. After that resulting combined database is sorted based on sorted key value, then sliding is performed. The central idea behind being, similar records come closer, after merging and sorting [16].

Candidate record pairs, which are produced in SNH approach is independent of the frequency of sorted key values, unlike BKVs in standard blocking.
2.2. Entity Resolution Process

For illustration, let,

\( n \) = size of two databases, whose records should be matched.

\( w \) = window size

Then, there will be \((n-w+1)\) window positions. Number of candidate pairs generated in first position is \( w \cdot w = w^2 \). Then in next each step \( w + (w-1) = 2w - 1 \) new pairs are added. If \( c \) is the total unique candidate records count, then \( c \) is,

\[
 c = w^2 + (n - w)(2w - 1) = 2nw - 2w^2 - n \tag{2.3}
\]

In case of many duplicate SKVs i.e., sliding window \( w \) fall less, then probable true matches will be ignored. An improved approach \[35\], which employs generation of inverted index data structure, as in case of standard blocking approach. The structure contains unique SKVs.

Another shortcoming of basic sorted neighbourhood approach is the fixed length of window size. This could result in ignoring of true match pairs, in case they are far away. To overcome this shortcoming, a more efficient method is proposed \[57\], where window size \( w \) is dynamically adopted, depending on underlying data distribution.

3. Q-Gram Based Indexing: Standard blocking and SNH blocking techniques are not efficient, when data are noisy and contain high errors percentages. They may not group true matching records into a respective blocks. In q-gram based indexing, for each variations of BKV, index keys are generated for each variation. Thereby, each record could be placed in more than one block, depending on variations of BKV \[18\]. Q-gram (also n-gram \[58\]) indexing approach transform each BKV(SKV) into list(substrings) of further q-grams. If \( q \) is ‘two’ then it is commonly called as bigrams or digrams \[59\]. If \( q \) is ‘three’, then called as trigrams \[60\]. A string \( s \) of length \( c \) and if \( k \) is number of possible q-grams, then \( k = c - q + 1 \) q-grams. For instance, the bigram list for the string ‘magdeburg’ is \(['ma', 'ag', 'gd', 'de', 'eb', 'bu', 'ur', 'rg']\).

The major shortcoming of q-gram based approach is that, huge number of sub-lists (even for small length of BKV) are produced. Hence, each record is most likely to get placed in many blocks. Thereby, it is computationally expensive, especially for long BKVs. Because of each record is inserted into many blocks, there by true matching of respective candidate rows are guaranteed, and this lead to high true match records and improved quality.

4. Suffix-Array Based Indexing: Suffix array based indexing is developed to improve the drawbacks of q-gram based approach. For understanding, suffixes are subset of substrings generated from a string. Suffixes are generated by removing each character from the beginning of a string. For instance, suffixes of ‘berlin’ are ‘erling’, ‘rlin’, ‘lin’, ‘in’ and ‘n’.

Inverted index is generated by adding each unique suffix as a key to the list. Each record that contains suffix as key will be inserted into the respective blocks\[61\]. Thereby, the probability of one record being inserted into more than one block exists.
Table 2.2: Suffix array based example with four names, modified from [64].

<table>
<thead>
<tr>
<th>RecordIDs</th>
<th>BKVs</th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>christina</td>
<td>christina, hristina, ristina, istina, stina, tina, ina</td>
</tr>
<tr>
<td>B</td>
<td>kristina</td>
<td>kristina, ristina, istina, stina, tina, ina</td>
</tr>
<tr>
<td>C</td>
<td>catherina</td>
<td>catherina, atherina, therina, herina, erina, rina, ina</td>
</tr>
<tr>
<td>D</td>
<td>katherina</td>
<td>katherina, atherina, therina, herina, erina, erina, rina, ina</td>
</tr>
</tbody>
</table>

Table 2.3: Suffix array based example with four names, modified from [64].

<table>
<thead>
<tr>
<th>Suffix</th>
<th>RecordID</th>
</tr>
</thead>
<tbody>
<tr>
<td>christina</td>
<td>A</td>
</tr>
<tr>
<td>hristina</td>
<td>A</td>
</tr>
<tr>
<td>ristina</td>
<td>A, B</td>
</tr>
<tr>
<td>istina</td>
<td>A, B</td>
</tr>
<tr>
<td>stina</td>
<td>A, B</td>
</tr>
<tr>
<td>tina</td>
<td>A, B</td>
</tr>
<tr>
<td>ina</td>
<td>A, B, C, D</td>
</tr>
<tr>
<td>kristina</td>
<td>B</td>
</tr>
<tr>
<td>catherina</td>
<td>C</td>
</tr>
<tr>
<td>atherina</td>
<td>C, D</td>
</tr>
<tr>
<td>atherina</td>
<td>C, D</td>
</tr>
<tr>
<td>therina</td>
<td>C, D</td>
</tr>
<tr>
<td>herina</td>
<td>C, D</td>
</tr>
<tr>
<td>erina</td>
<td>C, D</td>
</tr>
<tr>
<td>rina</td>
<td>C, D</td>
</tr>
</tbody>
</table>

This approach was employed in matching of English and Japanese bibliographic databases [61]. If \( c \) be number of candidate pairs, \( b_{\text{max}} \) is second parameter, \( b \) number of blocks generated, then estimated \( c \) can be obtained [35], in case of single database,

\[
c = b.b_{\text{max}}^2
\]  

(2.4)

When two databases are employed, then \( c \) is,

\[
c = b.(b_{\text{max}}(b_{\text{max}} - 1))/2
\]

(2.5)

5. Learning Optimal Blocking Keys: Traditionally, blocking keys are defined by respective domain experts for the above discussed blocking techniques. Evaluations [18, 35] show that manually defining optimum blocking keys for high data quality is not an easy task. The main purpose of finding optimal blocking is to reduce non-true match records as low as possible and to group all true matching records into blocks i.e. to increase quality and decrease the search space. Another way to compose the definition of blocking key is to learn from the underlying data in an automated way.
Automated machine learning algorithms [62, 63] are proposed based on supervised learning methodology. These algorithms needs training data of true-matches and non-true matches for learning purposes. Then, automatic blocking keys are obtained, the keys which have maximum coverage and maximum accuracy are selected.

2.2.3 Field and Record Comparison

Even after application of various advanced data cleaning techniques, resultant data may not be of high quality. This could be of various reasons discussed earlier like typographical variations of string fields, time-line changes and existence of various metaphorical forms. This problem persists mainly for attributes like ‘names’, ‘addresses’. Broadly, there are two types of comparison approaches [6], they are:

1. Exact comparison functions: In this, post comparison of attributes of candidate records, a binary value (‘match’ or ‘non-match’) is returned.
2. Approximate comparison functions: In this type, after comparison an estimation of similarity is returned.

For instance, \( s \) be the numerical similarity, \( a_i, a_j \) could be any attributes of string or numbers or ages or complex values or XML documents, then similarity function could be written as \( s = sim(a_i, a_j) \). Similarity value is always normalized i.e. \( 0 \leq s \leq 1 \). Below are the possible outcomes and explanation of them:

- If \( \text{sim}(a_i, a_j) = 1 \), then it means exactly similar
- If \( \text{sim}(a_i, a_j) = 0 \), then it means completely different
- If \( 0 < \text{sim}(a_i, a_j) < 1 \), then it means ‘somewhat similar’, this again need to be further classified and analyzed depending on data.

Next three comparison functions related to ER are discussed below.

- **Edit Distance String Comparison:** The least number of edit operations that are needed to transform from one string into another. There are,
  - **Insert:** To add any character(s), into the string.
  - **Delete:** To remove or delete any character(s), from the string.
  - **Replace:** To update or replace with any character(s) with in the string.

Dynamic programming algorithm [64] calculates the distance by counting, number of edits required to convert string \( \sigma_1 \) to \( \sigma_2 \). Time required for calculation : \( O(|\sigma_1| \times |\sigma_2|) \), \(|\sigma|\) denoting the length of string. Search space : \( O(min|\sigma_1|, |\sigma_2|) \)
• **Q-gram Based String Comparison:** Q-gram based string comparison is an approximate comparison approach. First step is to split the string into short characters of substrings of length $q$, called q-grams. The behind is when two strings are to be compared $\sigma_1$, $\sigma_2$, then they share many q-grams. Second step is to use a sliding a “window” along the characters of string $\sigma$. First q-gram is obtained by selecting characters from position one, second q-gram is obtained by selecting characters from position two.

Let $c$ be the number of q-grams generated from string $\sigma$, then $c = |\sigma| - q + 1$, $|\sigma|$ being length of string. Most commonly value of $q$ is set to two for matching names and addresses [59].

After q-grams are generated, similarity $s$ between $\sigma_1$ and $\sigma_2$ is computed based on number of q-grams in common for both strings. If $c_{\text{common}}$ represents common q-grams of strings $\sigma_1$ and $\sigma_2$, $c_1$ be number of q-grams in $|\sigma_1|$, $c_2$ be number of q-grams in $|\sigma_2|$ then normalized similarity value $0.0 \leq s \leq 1.0$ can be computed by one of the below equations [6]:

$$\text{Overlap coefficient : } sim_{\text{overlap}}(\sigma_1, \sigma_2) = \frac{c_{\text{common}}}{\min(|\sigma_1|, |\sigma_2|)}$$ (2.6)

$$\text{Jaccard coefficient : } sim_{\text{jaccard}}(\sigma_1, \sigma_2) = \frac{c_{\text{common}}}{c_1 + c_2 - c_{\text{common}}}$$ (2.7)

$$\text{Dice coefficient : } sim_{\text{dice}}(\sigma_1, \sigma_2) = \frac{2 \times c_{\text{common}}}{c_1 + c_2}$$ (2.8)

• **Jaro and Winkler String Comparison:** Matthew Jaro and Winkler have developed the approximate string comparisons [20, 65], which mainly focus on string token. Basic Jaro combination is based on the ideas of edit distance and q-gram comparison techniques. It is different in two ways. Firstly, it counts number of matching characters $c$, in the longest string. Secondly, it counts count of transpositions $t$ (swapping of two characters of first results in second, example ‘uk’ and ‘ku’) in group of common strings. Then Jaro similarity value can be obtained as,

$$\text{Jaro similarity value: } sim_{\text{jaro}}(\sigma_1, \sigma_2) = \frac{1}{3} \left( \frac{c}{|\sigma_1|} + \frac{c}{|\sigma_2|} + \frac{c - t}{c} \right)$$ (2.9)

### 2.2.4 Classification

The classification of comparison-pairs generated is based on similarity metrics obtained from comparison step. In this step, comparison-pairs after comparison step are to be classified or labeled as ‘matches’ or ‘non-matches’. The background idea is that match pairs are more likely to be same real-world entities. Two categories of classification namely threshold-based (Section 2.2.4) and rule-based (Section 2.2.4) are elucidated below.
2.2. Entity Resolution Process

a. Threshold-based Classification

In threshold-based classification, all the comparison-pairs are labeled as ‘matches’ and ‘non-matches’. A third category is also possible as ‘potential-match’, which is a state in between first labels. The categorization is based on the value obtained by summing up the similarity metrics values of each individual field comparison. The value obtained post summation of all is termed as ‘SimSum’ [6].

Let the threshold value be \( t \), which is used to label two classes (matches and non-matches) and \((a, b)\) be the comparison-pair. Then, classification can be seen as:

\[
\text{SimSum}(a, b) \geq t \Rightarrow (a, b) \Rightarrow \text{Match}
\]

\[
\text{SimSum}(a, b) < t \Rightarrow (a, b) \Rightarrow \text{Non-Match}
\]

In case of potential match class is considered. Then, \( t_l \) (lower limit) and \( t_u \) (upper limit) are needed and classification is shown below:

\[
t_l < \text{SimSum}(a, b) < t_u \Rightarrow (a, b) \Rightarrow \text{Non-Match}
\]

b. Rule-based Classification

In this type of classification, rules are used to categorize comparison-pairs into classes matches, non-matches and potential-matches [56]. A rule-based classifier is employed on similarity metric values computed from comparison (Section 2.2.3) step. Rules are formed by combinations of conjunctions (logical and \( \land \)), disjunctions (logical or \( \lor \)) and negations (logical not \( \neg \)).

2.2.5 Evaluation of Matching Quality

The last step of ER process is evaluation. All the previous steps are designed to achieve the maximum possible matching quality. In order to verify the quality of output after classification step, ground-truth data is required, also known as ‘gold-standard’ data [6]. The characteristics of the mentioned ground-truth data must assimilate data being matched. Normally, ground-data is not readily available for data matching projects. The followed approaches to acquire ground-data are: gathering results from previous works of the same domain, manual generation, use publicly available research test data. There is no real way to acquire (or generate) gold-standard ground-truth data, various combinations of above mentioned approaches are applied to fulfill the requirement.

After some acquirement of ground-truth data with true match status of comparison-pairs, each of the comparison-pairs is classified into one of the following four categories [66] and pictorial representation is shown in Figure 2.5:

- **True positives**: These are comparison-pairs, which are classified as matches by classification step (Section 2.2.4) and which are also indeed true matches. These entities are real-world similar objects.

- **False positives**: These are comparison-pairs, which are classified as matches by classification step and which are actually not true matches, also called as false matches. Classification step has been wrong.
• **True negatives:** These are comparison-pairs, which are classified as non-matches by classification step and which are *actually* non-matches. These entities are not real-world similar objects.

• **False negatives:** These are comparison-pairs, which are classified as non-matches by classification step and which are actually true matches, also called as false non-matches. These entities are actually refer to same real-world objects, but classification step has been wrong.

![Classification block diagram of record matching process](image)

The most common quality measures which are calculated based on true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) are precision, recall and F-measure. The characteristics and calculation are described below:

• **Precision:** It is commonly used quality parameter to estimate the quality matching process [54].

\[
\text{precision} = \frac{TP}{TP + FP} \tag{2.10}
\]

As computation of precision has no true negatives (TN) in it, thereby it is not affected by class imbalances.

• **Recall:** It is the second widely used quality measure in evaluation of data retrieval techniques [54]. It is also called as *true positive rate* or *hit rate*. Recall can be calculated as below:

\[
\text{recall} = \frac{TP}{TP + FN} \tag{2.11}
\]

As recall also does not contain true negatives (TN) in its calculation, it also immune to class imbalances, as accuracy. Recall calculates the fraction of true positives (TP) to that of true matches (TP + FN).
2.3 Data Skew and Load Balancing

- **F-measure**: It calculates the harmonic mean of precision and recall\(^67\), also called as \(f\)-score. It merges precision and recall.

\[
f - measure = \frac{precision \times recall}{precision + recall}
\]  

(2.12)

As from the equation, it is evident that f-measure will be high only when both recall and precision are also high.

### 2.3 Data Skew and Load Balancing

In general, when there is any join operation, the performance of the distributed system depends on the uniformity of key column. In the real world use case of traffic generated by communication towers, 80% of communication traffic is generated from 20% of the towers \(^68\). Even in the case transportation sector, for instance, in any city, only few points in the city will have maximum number of vehicles passing through, only a few airports in a country handles maximum number of traffic etc.

In particular to ER when the blocking technique is used, the major bottle necks will arise due to large sized blocks. This uneven distribution of block sizes is called as **data skew** in our context. Actually, the word **block skew** may be an apt word for data skew, but the reason for the block skew is, in first place data is skewed. Only because of data is skewed, it is translated to uneven block sizes. Therefore, the word **data skew** is used to pronounce this problem. In the following sections of this book, we refer this uneven distribution problem as **data skew**.

In the context of our work, the word **load-balancing** is used to address the problem created by data skew. As the larger blocks will create the bottle necks on any node (or core), to distribute this ‘bottle neck’ across is addressed as balancing of load. By only increased degree of balanced load, the efficiency and effectiveness of the underlying resources can be improved.

### 2.4 Distributed Computation Engines

In previous section, we have discussed the process of ER in detail. As ER is computationally and data-intensive, sequential programming model has hit it’s limits and ER process of large datasets is taking time up to few days to weeks \(^69\). In this context, distributed computing model has received attention to this kind of time-intensive tasks. Next, we introduce the popular distributed computing engines: Apache Hadoop (in Section 2.4.1), Apache Samza (in Section 2.4.2), and Apache Spark (in Section 2.4.3).

#### 2.4.1 Apache Hadoop

Apache Hadoop is an open source big data framework for distributed processing of prodigious amount of data across commodity computers in mostly transparent way \(^12\). It is the first open source big data framework to garner sufficient momentum in open source community. By design, it is highly scalable from a single node
to thousands of nodes in a cluster, in a share nothing architecture. Each node will provide local computation and storage. Hadoop relies on itself to handle any hardware failures. It is designed to detect and act w.r.t to hardware failures, which are common in any industrial scale installations.

**Hadoop Distributed File System (HDFS):** Any distributed files systems should deal with all the nuances related to network connectivity. Thus, making them complex compared with local file systems. To deal with such things, Hadoop has developed its own distributed filesystem called HDFS. It also has general purpose abstraction, to integrate with other file systems. It is designed to store very large files across the commodity type hardware with steaming access. It is designed for delivering high throughput, but at the cost relatively high-latency (not in the range of tens of milliseconds).

**Hadoop MapReduce:** MapReduce is the Hadoop’s programming model for data processing (batch processing engine). Hadoop can run MapReduce model programs written in programming languages like Java, Ruby and Python and by default all the programs run in parallel. To take full advantage of distributed computing, each query should be expressed as MapReduce job. Each job consists of two phases: map phase and reduce phase.

**Hadoop Yet Another Resource Negotiator (YARN):** Hadoop YARN is Hadoop’s coordinating part of the cluster. YARN stands in between of the application and making the underlying resource management and job scheduling in distributed environment transparent. It is introduced in Hadoop 2 to supplement MapReduce, but in general it is possible to run other diverse distributed paradigms (like Spark, Tez etc) as well.

### 2.4.2 Apache Samza

Samza is a simple, scalable, pluggable, fault tolerant big data stream processing framework with Apache Kafka and YARN at the core of its architecture. Kafka is part of it’s internal messaging module and YARN is responsible for fault tolerance, process isolation, security and resource management. In contract to MapReduce, it provides a simple mechanism of call back process-based APIs. It has features of snapshotting and restoration, when restarted to takes a consistent snapshot point. In the case hardware failure, in coordination with YARN transparency of data state is maintained. It can be seen as three layered system as below:

- **Streaming:** Apache Kafka is used here as the *distributed streaming platform* for the messaging module. It is used for construction of real-time stream pipelines and it is fault-tolerant, highly scalable and wickedly fast.

- **Execution:** YARN is used as a execution engine. It is already covered in Section 2.4.1.

- **Processing:** Samzas API is used as processing engine. It provides stream processing support for synchronous and asynchronous messages.
2.4. Distributed Computation Engines

Figure 2.6: Apache Samza architecture (left) along with Apache Hadoop architecture (right) [7].

Samza’s architecture is shown in Figure 2.6 along with Hadoop’s architecture in this regard. The main idea is that Samza’s architecture is flexible enough for plug and play. As can be seen, Kafka and YARN could be replaced with others if and when required.

2.4.3 Apache Spark

“Apache Spark is an open-source, high performance, massively parallel, in-memory execution environment, a general purpose distributed computing framework for large scale data processing with lightning fast analytics engine” [13]. In other words, it makes big data processing easy. It is the in-memory layer above the many data storages, into which data can be loaded to and analyzed in parallel. It can process data what normally a single computer can hold and beyond, with high level of transparency and relatively user friendly APIs. Spark was designed to cater for wide variety of workloads, which include batch applications, streaming, interactive queries, iterative algorithms. Previously more than one engine was required to deal these diverse workloads. The main features of Spark are:

**Speed:** With usage of state-of-the-art directed acyclic graph (DAG), physical execution engine, and a query optimizer Spark performs 100x faster than Hadoop [13].

**Ease of use:** Spark application can be quickly written in any one of the popular programming languages: Java, SQL, R, Python or Scala. It has over a set of 80 high-level operators to write parallel programming approach. All the Spark’s capabilities are accessible easily and quickly with data at scale, via a set of rich APIs.

**Generality:** Spark provides easy way to combine and interact with various libraries, which include SQL, Dataframes, MLlib and Spark streaming seamlessly. This adds power to applications written in Spark and provides easy extendability also.

**Runs everywhere:** Spark can run on various platforms, which include Apache Mesos, Hadoop, Kubernetes, standalone, or in the cloud infrastructure. It can also variety of data sources like HDFS, Apache HBase, Apache Cassandra, and Apache Hive.

**History of Spark**

Spark was born in 2009 in AMPLab mainly focusing on big data analytics, at the University of California, Berkeley. The motivation for this project is the inefficient
2. Background

Figure 2.7: Spark's components stack.
2.4. Distributed Computation Engines

<table>
<thead>
<tr>
<th>Feature</th>
<th>RDD</th>
<th>DataFrame</th>
<th>DataSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazily evaluated</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OOPs Supported</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compile time safety</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Interanl optimization provided</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.4: Features comparison of Sparks API.

performance of MapReduce w.r.t to iterative and interactive computing jobs. Hence, from the start main focus of Spark was speed, thereby bringing in ideas of in-memory processing and efficient fault recovery. It was first introduced in the paper "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center" by Benjamin Hindman and Matei Zaharia. By early 2010 it became an open source project. After picking up fast traction in open source community, it moved to Apache Software Foundation.

Spark APIs: RDD, DataFrame and DataSet

RDD, DataFrame and DataSet are the three Spark official data abstraction interfaces available from Spark 2.0 version. The time line of the introduction of the APIs in Spark versions are: RDD since Spark 1.0, DataFrame since Spark 1.3, and DataSet since Spark 1.6. In RDD based computations user should say, HOW the actions should be performed, in contrast for DataFrames and Datasets, only WHAT needs to done should be specified. By this, later two have a greater room for optimization is reserved with the Spark. Next we introduce briefly the key aspects of the aforesaid three APIs.

**RDD:** These are the building blocks of Spark, even the DataFrame and DataSet will be converted to RDDs under the hood for final computations. At the core RDDs are lazily evaluated, immutable distributed collections of objects, spread across the cluster. RDDs are strongly-typed collections of objects.

**DataFrame:** Like RDDs, DataFrames are also immutable distributed collection of objects, but they give us the table schema like view of the loaded dataset, analogous to the rows, columns in a table of any SQL based database. These are also lazily evaluated, but they have been boosted with features like custom memory management and optimized execution plans. These extra features gives a huge performance gain for DataFrame API. DataFrames are highly untyped collection of generic object of type Row.

**DataSets:** It is an extension of DataFrame and also combines the best of RDD and DataFrame. It is designed to have both characteristics: strongly-typed (like RDD) and untyped (like DataFrame). It has both compiler safety like RDD and performance boosting features of DataFrame.

Sparks Stack Overview

The current Apache stack consists of Spark Core and four other components, while Spark Core being the heart of Spark’s framework. The other four components are

---

4https://www.apache.org/
optimized to address four different use cases. Any typical Spark application requires Spark Core and at least one of the four components. These components are designed to operate seamlessly interoperable. As the four components are built on Spark’s core, the fastness and general-purpose nature of the Core powers higher-level components regarding various workloads.

There are benefits attached to the tight philosophy of integration of various components. Firstly, any changes are pushed to Core component, the effect automatically specular to the dependent components too. Secondly, one software will try to cater to the many requirement, without trying to maintain different softwares for each case. We can see this is as one stop - many solutions. Next, we introduce Spark’s components briefly and shown in Figure 2.7.

**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:** Spark SQL is the component that is for interaction with structured data. It also support integration with traditional and existing persistence frameworks such as Hive tables, JSON and Parquet, along with JDBC and ODBC connections and also to Apache Hive variant - Hive Query Language (HQL). It also opens up interfaces to interact with SQL from the applications on RDDs with Python, Scala and in Java. It brings together SQL queries and advanced algorithmic analytics to a single point of interaction, which is very unique of it’s kind.

**Spark Streaming:** Streaming component of Sparks facilitates the processing of live streams, examples include production servers logfiles, queues of status updates from media. It provides APIs to transform streams into RDD datatypes, thereby providing scalability, throughput and fault-tolerant features as Spark Core. It is flexible to integrate with Apache Flume (efficient for logfiles) and Apache Kafka (for distributed messaging).

**MLib (Machine learning library):** It is designed to work with machine learning type applications in the Spark ecosystem. It is the scalable machine learning library with inbuilt support to commonly used machine learning algorithms and utilities, which include regression, clustering, classification, hypothesis testing and also support for model evaluation and data import.

**GraphX:** GraphX was initially a separate project at UC Berkeley research work, then later on it was donated to Apache Software Foundation ⁵ and integrated with Spark’s ecosystem. It is the component for analysis and manipulations for graphs and graph-parallel computations. Like Spark SQL and Spark Streaming, it also support Spark RDD API, allowing to create distributed directed graphs with properties attached to each vertex and edge [13].

⁵https://www.apache.org/
Architecture of Spark

Spark follows master-worker architecture and a manager in between them. In this context, physical machines are called as nodes. The master node is called as driver node, which hosts driver process and workers are called as executors, which hosts executor process. These both processes can be hosted in single machine (horizontal cluster) or on separate machines (vertical cluster) or in mixed configuration. Block diagram representation of Spark architecture is shown in Figure 2.8.

![Figure 2.8: Spark’s architecture](image)

The driver process is bestowed for mainly three things: scheduling, analyzing and distributing tasks among the executor in the cluster, maintenance of Spark applications information, and play a correspondence role to users requests. Driver process stands at the heart of Spark application, which maintains all the necessary information through out the life cycle of the application. The executor process is responsible for two thing: execution of tasks delegated by driver process and reporting back the results or status to the driver node. The cluster manger is responsible for controlling of actual physical machines and resource allocations to the Spark application. This can be achieved by one of the already in place cluster managers, either YARN or Mesos or Spark’s standalone cluster manager. In practice, there could be more than one Spark application running in the cluster. The three different variations of running of Spark application are shown in Figure 2.9.

How Spark Works

In Spark, user programs are run on driver node, which are run in parallel by default. Large datasets that are partitioned across the cluster are represented as RDDs, which are stored in executors. The Sparks execution engine is responsible for the data distribution and computations, which is governed by the parameters set while submitting the Spark job. A detailed explanation of submitting Spark job is covered in Section 5.1.3.
2. Background

All the operations are performed on RDDs, which are called transformations. Spark evaluates all the transformations lazily only when really required but not performing as it encounter the operation. As per the design of RDD, all the RDDs are immutable, which means any transformation will result in a new RDD with the chain of transformations in it. Till the point of real execution, RDDs hold the meta-data of operations and transformation till that point.

**Actions:** The execution starts only when Spark encounters an action in the user program. An actions is the operation, which return something other than RDD. Examples of actions could be returning something to non-Spark environment or to bring data to the driver (operations like count, collect or savingToFile etc).

**DAG:** Actions triggers Spark to build directed acyclic graph (DAG) with the help of the meta-data collected about series of transformations. The sequence of steps realized is called as execution plan for that action. Lazy evaluation avoid the communication overhead with driver(and other nodes), otherwise may require multiple passes with partial data requests. As a side effect, Spark achieves fault-tolerant feature effectively, as meta-data of the RDDs re-construction is present in every RDD. Other systems with mutable and eager model evaluations had to exclusively log updates to the RDDs for the sake of re-construction, which will cost the performance. In contract, Spark’s RDDs have information about it’s lineage and also re-construction can be parallelized as well.

**Narrow vs. Wide Dependencies:** There are two important types of procedures of evaluations of transformations: transformations with narrow dependencies and transformations with wide dependencies. Understanding these dependencies become important in order to understand the background data movements in partitions, and in performance tuning. Narrow dependencies transformations are those whose dependencies can be executed from the known subset of arbitrary data without having to know the informations about other partitions. Each child RDD is dependent on at most one partition of parent RDD. Because of this, there will not be any shuffling involved in transformations execution. Example transformations are: filter, map, flatMap. In contrast, in wide range dependencies, child partition will depend of more than one parent RDD to perform the operation. Example operations include
sort, groupByKey, reduceByKey, join. In wide range dependencies, there exists shuffling.

**The Anatomy of a Spark Job:** As Spark follows lazy evaluation paradigm, it wouldn’t start evaluation until driver program hits action operation. For each action, Spark scheduler builds up an execution plan and starts a Spark job. In turn, each job is a group of stages, which are transformations of wide range dependencies to construct final RDD. Finally, each stage is set of tasks, which are actual parallel computations takes place on the executors. Each task will get scheduled to be performed on the cores of the executor machine. The pictorial representation of SparkApplication, jobs, stages, and tasks is shown in Figure 2.10.

![Figure 2.10: Anatomy of Spark's jobs execution](image)
3. Related Work

Entity Resolution (ER) is first presented in the area of public health in [70], for linkage of birth records, to draw relationships within a given population. ER is an important topic of research for over forty years [35]. [53] has formalized the proposal of ER and put forward the probabilistic decision based mathematical model. Next major contribution was from Winkler with the proposal of Expectation-Maximization [71]. All the approaches were focusing on improvement of accuracy of ER process. [20] proposed blocking mechanism to reduce the comparison space. This improves the time cost in ER, while only records pertaining to the subset (block with the same blocking-key) need to be compared. Christen [35] provided an elaborate survey of different blocking techniques and many other approaches have been proposed in surveys [40, 41, 72, 73].

3.1 ER with MapReduce

As databases were becoming increasingly large, and the advent of big data [74], ER became was dealing with huge amounts of data, which require massive computational power and storage capacities. At this juncture, researchers began to use distributed computing environments taking advantage of parallelism, to compute ER [75–79]. Parallelism provided another effective methodology to reduce the long running times of ER process. In recent past MapReduce framework became hugely popular in the context of parallel ER [80]. Kolb and Thor [4] discussed regarding the reduce phase skew and introduced two load balancing algorithms, which are BlockSplit, PairRange. In these, first datasets are profiled to know the distribution and then load balancing techniques are applied. An improved version of aforesaid load-balancing techniques, BlockSlicer [47] and MSBlockSlicer [81] are proposed.

The goal of the set-similarity join is also to find out duplicates or near-duplicates. Set-similarity joins on a single machine are covered in great detail in [82–85]. In the field of information retrieval across web contents, join operations between textual representations and relations are performed. An approximate join strategy is proposed in [86] in RDBMS. A comparative study of MapReduce and DBMS
paradigms for large-scale data has been done in [87]. Using MapReduce framework parallel set-similarity joins are proposed in [79, 88]. In [79], a detailed study of effective partitioning of large datasets, replication, controlling data in memory during join is covered extensively with respect to MapReduce framework.

As ER can also be seen as join operation, several techniques are proposed in this direction. In [89], the crucial implementation details of MapReduce joins namely repartition join, broadcast join, semi-join and per-split semi join are presented. The arbitrary joins(theta-joins) implementation with the simple randomized approach with minimal sampling is presented in [90]. In the age of InfiniBand connections, where links speed up to several gigabytes per second, zero-copy network communications are possible. In this context, parallel databases will encounter skewed data, which causes load imbalances. In [F.J], Flow-Join is introduced, which is a distributed join algorithm that also takes care of attribute skew. This employs small histogram based approximations to identify bottlenecks at runtime.

Some work also has been done towards ER as an application. Dedoop(Deduplication with Hadoop [91]) is an easy to use web-based application for ER of large datasets and also with rich visualization features. It supports a various selection of blocking techniques, similarity functions, and load-balancing strategies in combination with blocking techniques. An exclusive study of skew in MapReduce based applications is presented in [92]. In this work, authors have presented five sources of skew and their best practices for skew avoidance and their limitations too are discussed. In the paper [93], partitioning skew along with computation skew is investigated and also a locality and fairness aware algorithm LEEN is proposed. It improves the skew caused by huge data transfer during shuffle phase and fairness among data nodes. This approach is not scalable for very large datasets, as key group information is loaded into memory, which is not practical always.

In efforts to build scalable load balancing techniques, a sketch-based profiling is introduce in [94]. This approach stores the key size information in a sketch structure rather than full profile (which is not practical in large datasets with millions or billions of blocks). It introduced optimal sketch packing technique, which uses the sketch structure to find the keys distribution, hence providing a scalable approach. Further work in this direction is present in [94]. In this, two load-balancing algorithms cell block division and cell range division are presented, which work over sketch-based profiles and also handling data skew associated with the ER.

### 3.2 ER with Apache Spark

Not much work has been done w.r.t to ER with the framework Apache Spark and very sparse literature in particular with the challenge of data skew in ER. In [95], authors tried to improve the load-imbalance at the stage of intermediate shuffle process with splitting and combination algorithm, by using reservoir sampling. Here, an approximate method is used to find out intermediate keys skew. In [81], to improve load-balancing in the phase of pairs-generation, a variation of Sorted Neighborhood Method(SNM) is employed with RDDs and broadcast variables. This approach does not consider the blocks and their sizes. In our work, we focus on the load-imbalance caused by data skew on the Spark based cluster environment and adopt accurate block distribution approaches to improve the execution times.
4. Load Balancing Strategies for Spark-Based Entity Resolution

In Chapter 2, we have discussed the process of ER, importance of ER, challenges of ER, and the parallel computing engines. The challenges of ER can be seen as two faced problem: size of the data and skewness of the data. To address the size problem, we are using Spark, which is a big-data, in-memory computing framework. The second problem i.e. skewness is not automatically handled by using of any big-data framework. In this chapter, we address the handling of skewness of the given dataset. Next, we introduce the Spark based ER workflow (in Section 4.1), problem analysis in the cluster environment (in Section 4.2), and we present the approaches to address the problem of skewness (in Section 4.3).

4.1 Spark-based ER Workflow

As discussed in Section 2.1, the process and steps of ER, which is a sequential process. The sequential and traditional model of ER is taking extremely long running times when applied on large datasets (in the order of millions). Therefore, ER in distributed environment to parallelize ER process has gained significance momentum with the advent of MapReduce [1] parallel computing model. Various other distributed frameworks are presented in Section 2.4. In this work, we use Spark (introduced in Section 2.4.3) as a distributed computing framework.

Before going into details, we give a short overview of the structure of the dataset used in the following sections. In our work, we have used synthetically generated dataset of personal information with columns as given name, surname, gender etc., and each record in the dataset as an individual person. Sample view of records are shown in Table 4.1.

Next, in this section, we describe ER with Spark. As described in Section 2.2 the general flow of ER in detail, here we give our implementation details specific to Spark. The steps involved are: (i) creation of SparkContext (ii) loading dataset (iii) pre-processing (iv) blocking (v) comparison-pairs generation (vi) comparison and (vii) classification. Figure 4.1 illustrates the steps, which are discussed below:
Creation of SparkContext:

The first step of any Spark program is to create a SparkContext and it is the entrance point to the execution of Spark code. It will be instantiated on the drive node and lives throughout the lifetime of the Spark application.

Loading the Dataset:

After the creation of Sparkcontext, the dataset is loaded into Spark’s one of the abstract distributed APIs (ex: RDD). While loading, there is an option to provide, ‘number of partitions’ desired. If it is provided, Spark will do it, according to its internal optimization. At this stage, we refer the dataset as \( ds_a \).

Pre-processing:

Pre-processing is the starting point of ER. It could be also seen as an optional step, depending on the quality of the input dataset. Since, the datasets are synthetically generated, with the focus on load-balance into considerations, the quality of data is of high standards. As part of this step, we have removed any empty rows from the input dataset. After pre-processing \( ds_a \) is transformed to \( ds_b \).

Blocking:

Blocking is performed to decrease the search space of the subsequent matching process to a feasible extent. We have applied the standard blocking technique (introduce in 1). Blocking key is constructed as follows: Let, DoubleMetaphone (introduced in 2) of a column is represented as \( dm(\text{column name}) \). Procedure used to calculate blocking-key is mentioned below:

- DoubleMetaphone of two columns \( dm(\text{given name}), dm(\text{surname}) \) are computed at first and let the computed values be \( dm_{gn}, dm_{sm} \) respectively.

- The first two letters of \( dm_{gn}, dm_{sm} \) are extracted and concatenated.

Blocking-key = first_two_characters(\( dm_{gn} \)) + first_two_characters(\( dm_{sn} \)).

The calculated blocking-key value for every row is added as a new column to the dataset. This new column plays a significant role in further sections of this chapter. After blocking step, \( ds_b \) is transformed to \( ds_c \). The generated blocking-keys for the running example dataset is shown in Figure 4.2.
Figure 4.1: Flowchart of baseline approach of ER, as described in Section 4.1.
Comparison-pairs generation:

The next step after the blocking is comparison-pairs generation. After the generation of blocking-key, a new column is been added to the dataset. Thereby, we have already divided the dataset into as many blocks as unique blocking-keys generated. To imagine the blocks, assume we have sorted the current dataset w.r.t column blocking-key. Then, we can see dataset is divided into blocks, with each block having the same blocking-key. Also, if we perform groupby and get sum on blocking-key, we get the picture of data distribution (to some extent) and block sizes of the dataset. Usage and handling of block sizes are discussed in Section 4.3 in detail of this chapter.

Once the blocks are determined based on the blocking-key, the idea is to compare the records within the same blocks. For this, comparison pairs must be constructed to go to actual comparison process. To achieve this, a self-join is performed on the column blocking-key. Thereby, all the rows, which have the same blocking-key will be paired together.

Comparison:

The comparison is the fourth step in ER, which is the most expensive in terms of time consumption. In each comparison-pair, similar columns are compared. For instance, if A-B be a comparison-pair, then ‘given name’ of record A is compared against ‘given name’ of record B, so on and so forth.

For the comparison of numeric columns, absolute values are compared. If the similarity value obtained is 1 means similar, 0 means non-similar. In the case of a string column, we have employed JaroWinker (introduced in Section 2.2.3) is employed for computing similarity metric value. JaroWinker outputs a numeric value between 0 and 1.

In both cases, comparison metrics will be in numerics. In our work, we have rounded the decimal values to four digits. In figure Figure 4.3, first two columns are comparison-pair, and rest comparison scores of given name, surname, postcode, and city.

Classification:

Classification is the fifth step in the implementation of ER. In this step each comparison-pair is labeled as ‘match’ or ‘non-match’ and we have adopted threshold-based classification model (introduced in Section 2.2.4). Threshold-based classification is based

---

**Figure 4.2:** Blocking keys of the running example with five records.
on a threshold value, in which a numerical value $t$ is taken as the threshold. In comparison step, each comparison-pairs similarity metrics values (of all columns) are summed up, the summed up value is called as ‘total SimSum’. The total SimSum is compared with the threshold value, to categorize as either ‘matches’ or ‘non-matches’. Let $(a, b)$ be the comparison-pair and $t$ is taken as a threshold value.

\[
\text{total SimSum}(a, b) \geq t \Rightarrow (a, b) \Rightarrow \text{Match}
\]

\[
\text{total SimSum}(a, b) < t \Rightarrow (a, b) \Rightarrow \text{Non-Match}
\]

In Figure 4.4, ‘totalScore’ column refers to ‘total SimSum’ and $t$ is set as 7. Hence, we see the status according to the set threshold value. After classification results are saved to HDFS. This work flow is referred to the Basic approach in next subsequent sections.

### 4.2 Problem Analysis in the Cluster Environment

In this section, we analyze the load imbalances of the above-described approach, on the worker nodes and how it is affecting the response time.

The Basic approach is applied to dataset DS1. The structure of dataset is already discussed in Section 4.1. DS1 consists of $1.1 \times 10^6$ records, $4 \times 10^{11}$ comparisons, and 15409 blocks. For full details of datasets used, please refer to Figure 5.1.
Table 4.2: Execution and configuration details of the Basic approach.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs generated</td>
<td>2</td>
</tr>
<tr>
<td>Number of stages generated</td>
<td>4</td>
</tr>
<tr>
<td>Number of tasks generated of each stage</td>
<td>84</td>
</tr>
<tr>
<td>Number of blocks generated after blocking step</td>
<td>15409</td>
</tr>
<tr>
<td>Number of partitions used</td>
<td>84</td>
</tr>
<tr>
<td>Total execution time</td>
<td>424 sec</td>
</tr>
<tr>
<td>Dataset used</td>
<td>DS1</td>
</tr>
</tbody>
</table>

As discussed in Section 2.4.3, the anatomy of any Spark program, the number of jobs, stages, and tasks generated after applying Basic approach to DS1 is shown in Table 4.2.

Figure 4.5 represents, the time the taken for the execution of each task, this means 15409 blocks are converted to 84 groups and these groups are processed by 28 cores in the cluster. From Figure 5.1, we can see that for DS1, the largest 5% of blocks consists of 92% of records. This means records in the blocks are non-uniformly distributed. This non-uniform distribution is also called as skewed distribution. Thereby, we can see a high degree of non-uniformity w.r.t to execution times of 84 tasks. Execution times of largest and smallest task are 15s and 4.8min.

As data skew is dominating the response time, we adopt some of the strategies, which are required to improve the unbalanced block distribution. In next section, such strategies are described for the improvement of load balancing.
4.3 Load Balancing Strategies

In Chapter 3, we have introduced various load balancing strategies designed for MapReduce programming model. From the scientific literature, we can see various approaches to improve the load imbalances by reducing the block unequal distribution. In this work, we would like to focus on accurate block distribution approaches. Therefore, the choices are (i) BlockSplit [4] (ii) Pair Range [4] (iii) MaxBlock (iv) BlockSlicer [96] and (iv) MSBlockSlicer [81]. All these approaches are presented in the order in which they had evolved in MapReduce programming model. We have adopted BlockSplit and PairRange because these are the core ideas in the category of accurate block distribution approaches. These approaches have been adaptations to the Spark programming model.

To implement any accurate block distribution approaches, first we need to have information about the number of partitions of the given dataset, number of blocks, and distribution of blocks in the partitions. The data structure which holds this information is referred as block distribution matrix. Next, in this section, we describe the computation of block distribution matrix (in Section 4.3.1), BlockSplit approach (in Section 4.3.2) and PairRange approach (in Section 4.3.3).

### 4.3.1 Computation of Block Distribution Matrix

Block distribution matrix (BDM) is a data structure used to hold the information about the number of records in each block of each partition. BDM can be seen as a $b \times p$ matrix, with $b$ being the number of blocks (generated after blocking step) and $p$ is the number of partitions. For illustration purposes, we consider the sample dataset shown in Table 4.3 consists of 14 records with four blocking keys.

#### Calculation of BDM

Calculate of BDM is a three-step process, as described below:

<table>
<thead>
<tr>
<th>Partition</th>
<th>Record</th>
<th>Blocking Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>A</td>
<td>AF0R</td>
</tr>
<tr>
<td>P0</td>
<td>B</td>
<td>AF0R</td>
</tr>
<tr>
<td>P0</td>
<td>C</td>
<td>AAT</td>
</tr>
<tr>
<td>P0</td>
<td>D</td>
<td>AAT</td>
</tr>
<tr>
<td>P0</td>
<td>E</td>
<td>AAT</td>
</tr>
<tr>
<td>P0</td>
<td>F</td>
<td>AFAF</td>
</tr>
<tr>
<td>P0</td>
<td>G</td>
<td>AFAF</td>
</tr>
<tr>
<td>P0</td>
<td>H</td>
<td>AF0R</td>
</tr>
<tr>
<td>P0</td>
<td>I</td>
<td>AF0R</td>
</tr>
<tr>
<td>P0</td>
<td>K</td>
<td>AAP</td>
</tr>
<tr>
<td>P0</td>
<td>L</td>
<td>AAP</td>
</tr>
<tr>
<td>P0</td>
<td>M</td>
<td>AFAF</td>
</tr>
<tr>
<td>P0</td>
<td>N</td>
<td>AFAF</td>
</tr>
<tr>
<td>P0</td>
<td>O</td>
<td>AFAF</td>
</tr>
</tbody>
</table>

Table 4.3: Dataset with partitions and with blocking key.
Figure 4.6: Block Distribution Matrix computations steps, adapted from [4].
4.3. Load Balancing Strategies

- An additional column named ‘Value*’ with value as ‘1’ is added to each record in the dataset.

- The blocking-key of every record is appended with respective ‘partition_id’. In our approach, we used ‘#' as a separator between blocking-key and partition_id. For instance, record A’s blocking-key is AFOR is updated to AFOR#0, H’s blocking-key (AFOR) is updated to AFOR#1.

- Dataset is grouped by blocking-key and is aggregated by Value*. Figure 4.6 represents the flow to calculate BDM.

\[
\begin{array}{|c|c|c|}
\hline
\text{Blocks} & \Phi_0 & \Phi_1 & \text{Computations/Block} \\
\hline
\text{AFOR} & 2 & 2 & 6 \\
\text{AFAF} & 2 & 3 & 10 \\
\text{AAP} & 0 & 2 & 1 \\
\text{AAT} & 3 & 0 & 3 \\
\hline
\end{array}
\]

Figure 4.7: Block Distribution Matrix.

Figure 4.7 represents the visual representation of BDM for the running example. It consists of four rows, as the number of blocks ($\Phi_0$, $\Phi_1$, $\Phi_2$, $\Phi_3$). The first two columns are two partitions ($P_0$, $P_1$). The third column is computations required for each block. Sizes of blocks range from two to five records. Comparisons per block range from 1 to 10 (calculated according to equation Equation 4.3). The total number of comparisons in the example dataset of 14 records is 20. In this 20 comparisons, smallest block($\Phi_2$) contributes 1 comparison and largest block($\Phi_1$) contributes 10 comparisons. It means, largest block and smallest block are contributing 50% and 5% of the total workload respectively.

4.3.2 BlockSplit Based Load Balancing [4]

In this approach, each block (records with the same blocking-key) is assigned with a match-task ($m_t$). Thereby, we gain the control of execution of blocks. Each match-task, in the Spark terminology, will become a part of the task (remember jobs $\rightarrow$ stages $\rightarrow$ tasks). The number of match-tasks will be taken as an input parameter for this approach. Each match-task can be seen as a logical unit of computation. From BDM and for a given match-task, comparisons per match-task can be computed (as $\chi$). Also, from BDM comparison required for each blocking-key can be computed. Then, there exist two categories of blocks: (a) blocks with comparisons less than $\chi$, (b) blocks with comparisons greater than $\chi$.

In the case (a), all the records of a blocking-key will be assigned with the single match-task. In the case (b), records in a partition are split $p$ times, giving rise to one original block and $p$-1 additional blocks. And each $p$-1 additional blocks are assigned with a same match-task, thereby not loosing all the required comparisons as the Cartesian product.
As per above idea, the load of larger blocks' comparisons are distributed across the match-tasks. To achieve this, a compositional key is key is required for each record. Hence, this approach is about the generation of compositional_key-value pairs, with key \(\text{match}\_\text{task}\_\text{index} \otimes \text{blocking-key} \otimes \text{split}\_\text{index}\) and value as the record itself. Then, grouping is performed on the key and the records in the newly formed groups are compared. Anatomy of the key is described below:

- The match_task_index value is assigned between 0 to \(m_t - 1\). It is used to redistribute the comparisons.

- Blocking-key value is a string, generated after blocking step. It is used to group all the records, which belong to the same blocking-key.

- The split_index is assigned between 0 to \(|p| - 1\). This is used to know whether an record is split or not.

Each record will have a blocking-key, which is generated after blocking step. Once, blocking-key is in place, the split_index can also be emulated, as we will also the partition information. In order to construct the compositional-key, which also match_task_index, we need to perform some computations to calculate the match_task_index for the combination of ‘blocking-key’ and ‘split_index’. Therefore, first a data structure ‘match_task_index map (MTIM)’ is computed (described in Section 4.3.2). Secondly, with the help of MTIM, a compositional-key is generated for every record (described in Section 4.3.2). Lastly, grouping on compositional-key is done and comparison pairs are generated (described in Section 4.3.2). Next, the three steps are described in the mentioned order.

(i) Computation of MTIM:

MTIM is a data structure to store the information about: the blocks, respective comparisons of block and respective match-task. MTIM hold the match_task_index for all the possible blocks and sub-blocks. The blocks are the consequence of blocking step and sub-blocks are the consequence of splitting of larger blocks, which exceed a threshold value (calculated as Equation 4.2).

Once BDM is initialized, total comparisons \(P\) of all the blocks are calculated as follows:

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

(4.1)

Next, the average number of comparisons \(\chi\) for each batch \((m_t)\) is calculated as below:

\[
\chi = \frac{P}{m_t}
\]  

(4.2)

Next, the comparisons for each block \(\epsilon\) is calculated as below:

\[
\epsilon = \frac{1}{2} \cdot |\Phi_k| \cdot (|\Phi_k| - 1)
\]  

(4.3)
4.3. Load Balancing Strategies

After calculation of $\epsilon$, two possibilities arise as shown below:

**case a:** If $\epsilon < \chi$, then the respective block is not further divided, it is grouped in a single match_task. The split_index is denoted as $k_*$. Sample entries in this category looks as $[(AAT.0._*=3), (AAP.0._*=1)]$, as blocking keys AAT, AAP does not exceed the average value $\chi$.

**case b:** If $\epsilon \geq \chi$, then the respective block is split into $p$ sub-blocks. This division leads to $\frac{1}{2}.p.(p-1) + p$ match-tasks. The first part $\frac{1}{2}.p.(p-1)$ comparison tasks, represented with $k.i \times j$ and $i,j \in [0, p-1]$ and $i < j$. Second part, $p$ tasks, represented as $k.i$, processing $i^{th}$ sub-block $i \in [0, p-1]$. The blocks, which belong to this category, will have $p$ entries in the MTIM, with split_index = $(i \times j)$ where $i,j \in [0, p - 1], i < j$. Sample entries in this category looks as $[(AF,1,0=6), (AF,1,1=3)]$.

After the key-value calculations, the whole structure is sorted by value in descending order. Next, each row’s value (i.e. comparisons) is replaced with a match_task from 0 to $m_t-1$ in a sequential fashion. Assigned match_tasks is denoted as $C(k.x)$. Sample entries looks like $[(AAP,0,0=2), (AF,1,1=1)]$, where values 1, 2 are match_task_index. Before assignment of match_task_index sorting of descending is done to make sure that largest chunks of comparisons are not assigned with same match_task.

**(ii) Computing compositional key-value pairs:**

The second step in the BlockSplit approach is computing key-value pair(s), for each record. From MTIM, we can extract match_tasks_index if we have a blocking-key and split_index. For the given blocking-key, we can construct the split_index for every record. Once the split_index is computed, from MTIM, we get a match_task_index. Thereby, the key is constructed as the $match_task\_index, blocking-key, split\_index$ and value = record.

In case, the number of comparison of the block exceed $\chi$ then, we output $p$ number of key-value pairs. One record with key=$C(k.i).k.i$ and remaining $p-1$ with key = $C(k.i).k.i \times j$, where $j \in [0, p - 1]$ and $j \neq i$. The $p-1$ split records are annotated with $i \times j$, which is used in next stages, to avoid multiple comparisons of same record pairs.

**(iii) Grouping and comparison-pair generation:**

The third step in computing BlockSplit is grouping (keys) and comparison (records in groups generated). After the generation of key-value pairs (in step Section 4.3.2), key-value pairs are grouped by key. Keys which have same ‘match_task_index, blocking-key, split_index’ are put into groups. Blocking-keys whose total comparison don’t exceed $\chi$ will be grouped into a single group and other which exceed $\chi$ will be split into $p$ groups.

After grouping, records within the group are compared. In construction of record pairs for comparison, the records, which are being duplicated (i.e., keys with suffix $k.i \times j$) are not been considered among themselves because of records with suffix $k.i$, will be compared anyway, hence avoiding duplicate record-pair comparisons.
Figure 4.8: Flowchart of the BlockSplit approach.
Figure 4.9: The BlockSplit approach work flow.
Illustration of BlockSplit for running example

BlockSplit approach for the running example of 14 records (Table 4.3) is described as follows:

- In the running example, \( P = 20 \) (calculated with Equation 4.1) and the number of match-tasks \( m_t=3 \), is taken as input. Hence, average comparison per core \( \chi = 6 \) (calculated from Equation 4.2).

- Referring to BDM Figure 4.7, only the block with blocking-key=AFAF(\( \Phi_1 \)) is above the \( \chi \). In partition zero \( P_0 \), \( \Phi_1 \) has two records (F and G), and these records will be split two (since \( p=2 \)) times. First set (F and G) of split will have key suffix as AFAF.0 and second set (\( F_0 \) and \( G_0 \)) will have AFAF.0×1.

- Similarly, in the second partition (\( P_1 \)), \( \Phi_1 \) only three records (M, N, and O). Therefore, only these three records will be split twice. The first set (M, N and O) will have key as AFAF.1 and second set (\( M_1 \), \( N_1 \), and \( O_1 \)) will have key suffix as AFAF.0×1.

- Hence, total 10 comparisons of block \( \Phi_1 \) are divided among match_task-0 (AFAF.0×1): 6, match_task-1 (AFAF.1): 3 and match_task-2 (AFAF.0): 1.

- Referring to BDM Figure 4.7, blocks \( \Phi_0 \), \( \Phi_2 \) and \( \Phi_3 \) do not exceed \( \chi \). Therefore, all the records of two partitions (except ‘F’, ‘G’, ‘M’, ‘N’ and ‘O’ ) represents only one row in Key-Value step (step Section 4.3.2). These records combined have 10 comparisons, which are distributed across match_task-0, match_task-1 and match_task-2.

- In effect 20 comparisons are distributed among 3 match-tasks as 9, 7 and 4 respectively. Thereby, load-distribution can be seen as 45%, 35%, and 20%.

4.3.3 PairRange Based Load Balancing [4]

In BlockSplit strategy, large blocks are split into as many times as input number of partitions. This improves the load-balance compared to Basic approach but, still suffers challenges regarding blocks generation. The main drawback is non-uniform sizes of sub-blocks. This is evident from the running example with 20 comparisons being distributed as 9(45%), 7(35%) and 4(20%). The difference of highest and lowest workload is 15% (i.e., 45% - 20%), which is not totally uniform distribution of load across the parallel batches. To improve this further, another approach PairRange [4] is employed. The PairRange approach follows virtual indexing of all records, comparison-pairs and grouping comparison-pairs into ranges.

Even in PairRange, the core idea is to construct a compositional key for every record, then group the dataset with compositional key and compare the records in the blocks generated after grouping. The compositional key is generated in such a way that all the possible record comparison are not lost. The structure of compositional key is range_index \( \otimes \) block_index \( \otimes \) entity_index. Firstly, with the help of blocking-key, entity_index is calculated (described in Section 4.3.3). Next, to compute the
range_index for a given record: indexing of comparison-pairs is required (described in Section 4.3.3) and ranges computation is required (described in Section 4.3.3). Next, with entity_index and range_index in hand, a compositional key is generated for each record (described in Section 4.3.3). Lastly in this approach, grouping on compositional key and comparison is done (described in Section 4.3.3).

Next, in this section all the five steps mentioned above are described. Then, the programmatic approach of PairRange is described in Section 4.3.3. Lastly, an illustration of PairRange with the running example is describe in Section 4.3.3.

(i) Virtual indexing of records:

The fist step in PairRange approach is virtual indexing (numbering) of records. Pre-calculated BDM is used in this step, of indexing records. Each block with $|\Phi_k|$ records can be arranged as matrix $|\Phi_k| \times |\Phi_k|$. Records of the same block are arranged as both columns and rows, thereby rank of the matrix will be $|\Phi_k|$. In this kind of arrangement virtual indexing is performed. Figure 4.10 represents the pictorial representation of the concept of numbering. Each matrix structure consists of numbering represent the indexing of records (entity_index), which can be seen at the upper and left periphery of the matrix. Also, the records, which belong to a single block will be part of more than one partition. Hence, continuous numbering is not a straightforward task. Next, we describe the procedure to achieve continuous numbering and procedure of computing entity_index.

![Figure 4.10: Enumeration of all pairs of running example, adapted from [4]. Three ranges are indicated with three different color coding.](image)

**Entity_index:** Entity_index is a sequential number (index) assigned to each record in a single block($\Phi_k$). Here, the context of sequential is important because all records of a block may not be present in a single partition. Even though records are distributed across the partitions, the sequential numbering is achieved with the help of BDM. From BDM, we can know the partition number, in which the record resides. Also, from BDM we can calculate the count of records, which are residing in the previous partitions of the respective block. This calculated value is the off-set of the record, which is being processed. Thereby, we can add the off-set to the local index (in partition $P_i$) to get the global index of the record in the whole dataset. The global index (entity_index) is possible to achieve, even though the system can process individual partition independently (or parallelly).
Let, given an record \( t \) belongs to partition \( P_i \), block \( \Phi_k \). Then, the preceding partitions \( P_0 \) through \( P_{i-1} \) has to be added to the local index of the block(\( \Phi_k \)). For instance, if we consider the record O, which is part of the second partition (\( P_1 \)) and block AFAF (\( \Phi_3 \)). The local index of O within the block(\( \Phi_3 \)) is 3. From BDM (refer Figure 4.7) we can infer (in practice we calculate) that in partition \( P_0 \) there are 2 records, which also belong to block AFAF. Hence, we can add up count (not index) from \( P_0 \) i.e., 2 and local index (not count) i.e., 2 to arrive at entity_index (global index) of ‘O’ as 4. All the calculated entity_index of the running example can see seen in Figure 4.10 (top and left margins).

(ii) Virtual indexing of comparison-pairs:

The second step in PairRange approach is virtual indexing of comparison-pairs (pair_index). BDM is used in this step of indexing of comparison-pairs as well.

**Pair_index:** Computation of entity_index is a prerequisite for the calculation of pair_index. Since each block can be seen as a matrix with rank as \( |\Phi_k| \), cell \( (a, b) \) can be considered as a pair. In cell \( (a, b) \), ‘a’ represents column index and ‘b’ represents row index of matrix of block \( \Phi_k \), where \( a, b < |\Phi_k| \) and \( a, b \in [0, |\Phi_k|-1] \).

**Significance of entity_index in pair-comparisons:** Entity_index plays a significant role in the process of the pairwise comparison process. Let, say pair \( (a, b) \) with ‘a’, ‘b’ being entity_index of the two records and the pair \( (a, b) \) is considered for comparison only if \( a < b \).

- This check will avoid the unnecessary computations that both records being same i.e., \( (a,a) \) or \( (b,b) \).
- This also avoids redundant computation of \( (b, a) \).

**Indexing scheme:** From BDM, list of blocks along with their sizes are known. Starting from first (in any order) numbering is performed, as follow:

- Pairwise indexing (numbering) follow column-wise scheme i.e., starts at the first column and goes from top to bottom, then the next second column and so on. In this defined path, sequential numbers are assigned to all the cells in a matrix (or block \( \Phi_k \)), which satisfy the condition \( a < b \).
- Next, the numbering process continues for the next block (from the BDM list) in a continuous manner. Thereby, every eligible pair-comparison will be numbered in the sequence of blocks considered from BDM.

Next, how to calculate the index for each cell is described.

**Calculation of pair_index of every cell:** Let pair_index of a pair \( (a, b) \) be denoted as \( p_i(a, b) \) where \( a < b, a, b \in [0, |\Phi_i| - 1] \). Then \( p_i \) can be calculated as follows:

\[
p_i(a, b) = g(a, b, |\Phi_i|) + h(i)
\]  
(4.4)
4.3. Load Balancing Strategies

calculation of \( g(a, b, |\Phi_i|) \): \( |\Phi_i| \) denotes the number of records in the respective block. This could be the count of records of block \( \Phi_i \) or rank of the matrix with block \( \Phi_i \) and it is denoted as a natural number \( N, N \in \mathbb{Z}^+ \). Here, \( g(a, b, N) \) is the index of the cell \( g(a,b) \) in the matrix of \( N \times N \). Calculation of \( g \) is given as, 
\[
g(a, b, N) = \frac{a}{2}(2N - a - 3) + b - 1
\]

calculation of \( h(i) \): It denotes the offset w.r.t to the all the previous blocks of \( \Phi_i \) with \( \Phi_0 \) to \( \Phi_{i-1} \). Calculation of \( h \) is given as, 
\[
h(i) = \frac{1}{2} \sum_{k=0}^{i-1} (|\Phi_k| - (|\Phi_k| - 1))
\]

In the running example of 14 records (Table 4.3) with \( P = 20 \), \( \text{pair\_index} \) is numbered from 0 to 19. \( \text{Pair\_index} \) is running through four blocks (\( \Phi_0, \Phi_1, \Phi_2 \) and \( \Phi_3 \)) in a continuous manner. \( \text{Pair\_index} \) can also be seen as an aggregation of comparisons within blocks, progressively. For instance, \( \text{pair\_index} \) of \( p_i(2,4) \) with 2 as column index and 4 as row index, of block \( \Phi_1(AFAF) \) is 14. Figure 4.10 shows the \( \text{pair\_index} \) of all the eligible pairs \( p_i(a,b) \), where \( a < b \).

(iii) Range calculation:

After virtual indexing of all records and pairs, \textit{PairRange} approach splits the \( \text{pairs\_index} \) into to almost equal ranges. Let, say there are \( \Gamma \) ranges (taken as input) from \( \Gamma_0 \) to \( \Gamma_{r-1} \) for the calculated \( P \). The effort is to assign \( k^{th} \) range \( \Gamma_k \) to the \( k^{th} \) core. For the given, \( P \) total pairs, \( r \) ranges, a pair with pair\_index of \( p \) where \( 0 \leq p < P \) belongs to \( \Gamma_k \) by the below condition:

\[
p \in \Gamma_k \iff k = \left[ \frac{r \cdot p}{P} \right]
\]

Since we divide total pairs \( P \) into equal ranges, therefore last range will have unequal pairs, the leftover of \( (r-1) \) ranges.

In the running example of \( P=20, \Gamma=3 \), we get the ranges \( \Gamma_0 = [0,6] \) and \( |\Gamma_0|=7, \Gamma_1 = [7,13] \) and \( |\Gamma_1|=7 \) and \( \Gamma_2 = [14,19] \) and \( |\Gamma_2|=6 \). Different ranges in Figure 4.10 are shown with different color coding from 0 to 19 across the four blocks. As the size of the blocks is dependent on the size of ranges, therefore \( |\Gamma| \) is used to control the size of blocks (which is taken as input for the approach).

(iv) Key-value pair generation:

The fourth step in \textit{PairRange} approach is key-value pair(s) generation for each record in the dataset. Computations from previous steps, namely entity\_index, range\_index and BDM are utilized in this step. For each record, key = \( \text{(range\_index} \otimes \text{block\_index} \otimes \text{entity\_index}) \) and the value = record itself is generated.

Key generation: Procedure to construct a key for an record is explained below:

- Block\_index can be taken from BDM as every record posses blocking key.
- Entity\_index is calculated (as described in Section 4.3.3) from BDM.
- All the eligible pairs \( (a, b, a < b) \), which have respective entity\_index are constructed and pair\_index of all the pairs are computed.
- Range\_index (could be more than one) is calculated described in Section 4.3.3. Hence, every record may have more than one range\_index i.e., \( \Gamma_k \).
Figure 4.11: Flowchart of the PairRange approach.
(v) Grouping and comparison-pairs generation:

The fifth step in PairRange approach is the grouping of the key-value pairs constructed in the above step (Section 4.3.3). As the key part consists of range_index, block_index and entity_index, grouping is performed only on range_index and block_index. Thereby, records belong to same range_index are directed to be processed as a single group. This single unit could be sub-divided due to the presence of block_index. Finally, all the groups (or sub-groups) could be processed parallelly and independently.

After grouping is performed comparison-pairs are computed from groups generated from the above step. Normally Cartesian product must be performed in the groups or sub-groups generated. In this context, the Cartesian product will lead to unnecessary and redundant. Hence, all the pair combinations not are considered for comparison in a group.

The Programmatic approach of PairRange:

In this part, we describe how the above-mentioned approach, from step(i) to step(v) can be programmatically realized:

- As the first step, BDM is initialized and $P$ is computed.
- As an input parameter number of ranges is read, and the respective number of ranges ($\Gamma$) is determined.
- Each record $t$ from each partition is read and its entity_index $x$ is determined (as described in Section 4.3.3).
- Determining ranges: Since each record could be part of more than one range, all the ranges to which the record belong to are determined. For instance record N from the Figure 4.10 we can see that it belongs to two ranges (two different color codings). Therefore, record N belong to range one ($\Gamma_1$) and range two ($\Gamma_2$).
- From BDM, size of the block $N$ is determined. Then $t$ will be in ranges, in which the following pairs are in: $(0, x)$, $(1, x)$,...,$(x-2, x)$, $(x-1, x)$,...,$(x, N-2)$ and $(x, N-1)$.
- Thereby, determining only first and last range is good enough to get them in between ranges, i.e., $p_{min}$ and $p_{max}$ pertaining to $(0, x)$ and $(x, N-1)$ respectively. Since, ranges are increasing with pair_range (as seen in Equation 4.5), determining the first and last i.e., $p_{min}$ and $p_{max}$ are enough.
- After determining the range(s) for each record, key-value pair(s) are generated for each record. If there exists more than one range, multiple key-value pairs are produced for one record. Key is structured as range_index $\otimes$ block_index $\otimes$ entity_index and value = $t$ (i.e., record itself).
- Then, grouping is done on first two part of the key i.e., range_index.block_index.
As a result, blocks which belong to same range and block are produced.
Figure 4.12: Flow of PairRange working example.
Next, records in the respective block are compared for the similarity check. Not every combination of the pairs in the blocks are compared against each other, only if the following two conditions are satisfied: (i) Pairs which belong to the range of the respective range_index and (ii) first record’s entity_index should be less than second record’s entity_index. These extra checks are required to avoid duplicate and redundant checks.

**Illustration of PairRange with running example:**

For the running example, $\Gamma$ is taken as 3 and $P = 20$ comparisons. Then $[0, 19]$ comparisons must be divided into three ranges. Hence, three ranges we get as $\Gamma_0 = [0, 5]$, $\Gamma_1 = [6, 11]$ and $\Gamma_2 = [12, 19]$. First, two ranges have similar range i.e., $|\Gamma_0| = |\Gamma_1| = 6$ and last range ($|\Gamma_2|$) has 8. Then, corresponding ranges are determined for every pair_index. All the records except M, N, O, and F belong to the only one range($\Gamma$). In Figure 4.10, it can be seen that only these (four) records have two color combinations(row wise or column wise). Hence, all these records output two rows of key-value pair and other output only one key-value pair (seen in Figure 4.12 in: A. Key Generation step).

For instance, G is part of block $\Phi_1$ (AFAF) with entity_index 1 and pair_indices as 6, 10, 11 and 12. The pair_indices $[6,10,11] \in \Gamma_1$ and $[12] \in \Gamma_2$, hence two pairs are generated as $[1.1.1, G]$ and $[2.1.1, G]$. Normally, G should be compared against every record whose entity_index is greater than G(i.e., 1). Since, first range will only process the pair_indices in the range $[6, 11]$, the combination of ‘G-O’ is not considered as it’s pair_index is ‘12’(also shown for every pair in Figure 4.12), which is not part of $\Gamma = [6,11]$.

In this example, we can see that $\Phi_1$ (AFAF) records are spread over two ranges namely $\Gamma = 1$ and $\Gamma = 2$, thereby splitting the loading across two logical units. Similarly, $\Phi_0$ (AFOR) is processed by range $\Gamma = 0$ and $\Phi_3$ and $\Phi_3$ are processed by core $\Gamma = 2$. The total load of 20 comparisons is divided across three units 6, 6 and 8 respectively. First two units take the equal load and third is been given a slightly high (2 extra computations) load. In the end, the load is distributed as 30%, 30%, and 40% respectively.
4. Load Balancing Strategies for Spark-Based Entity Resolution
5. Evaluation

In Chapter 4, we covered the theoretical and practical approaches of two load-balancing strategies i.e., BlockSplit and PairRange. In this chapter, we evaluate these strategies regarding two performance-critical factors: robustness and scalability. In Section 5.1, we describe the experimental setup, software and hardware specifications. In Section 5.2, we describe the tuning of three approaches. In Section 5.3, we describe tests regarding effect of skew factor. In Section 5.4, we describe our third experiment regarding the speed-up.

5.1 Experimental Setup

In this section, we describe the environment that was used to conduct our experiments. Next, we describe the datasets used for evaluation (Section 5.1.1), experiments running environment details (Section 5.1.2) and details of spark job submission (Section 5.1.3).

5.1.1 Datasets

In our work, we used FEBRL [76], which is an open source data generator, a python based software developed as part of FEBRL project by Peter Christen. Since it is difficult to acquire gold standard data with varying sizes and varying skew, we used the FEBRL to generate synthetic datasets for our experiments. Information about datasets, which are going to use for evaluation is presented in Figure 5.1.
### Dataset used for tuning the three approaches

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>#Entities</th>
<th>#Comparisons</th>
<th>Number of blocks</th>
<th>Largest 5% of blocks statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>119 MB</td>
<td>$1.1 \times 10^7$</td>
<td>$4 \times 10^{11}$</td>
<td>15409</td>
<td>50%</td>
</tr>
</tbody>
</table>

### Datasets used for testing robustness

<table>
<thead>
<tr>
<th>Dataset Size in MB</th>
<th>#Entities</th>
<th>0%</th>
<th>20%</th>
<th>40%</th>
<th>60%</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TNC</td>
<td>LBC</td>
<td>TNC</td>
<td>LBC</td>
<td>TNC</td>
<td>LBC</td>
</tr>
<tr>
<td>Type I DS2</td>
<td>1.12</td>
<td>$1 \times 10^4$</td>
<td>0</td>
<td>$2 \times 10^5$</td>
<td></td>
<td>$8 \times 10^6$</td>
<td></td>
</tr>
<tr>
<td>Type II DS3</td>
<td>1.13</td>
<td>$1 \times 10^4$</td>
<td>$0.5 \times 10^5$</td>
<td>$7.3 \times 10^5$</td>
<td></td>
<td>$2.3 \times 10^6$</td>
<td></td>
</tr>
</tbody>
</table>

TNC : Total Number of Comparisons  
LBC : Largest Block’s Comparisons

### Dataset used for testing speedup

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>#Entities</th>
<th>#Comparisons</th>
<th>Number of blocks</th>
<th>Largest blocks statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS4</td>
<td>30MB</td>
<td>$3 \times 10^5$</td>
<td>$1.8 \times 10^5$</td>
<td>12814</td>
<td>1%</td>
</tr>
</tbody>
</table>

Figure 5.1: Test datasets used for evaluation.
5.1. Experimental Setup

<table>
<thead>
<tr>
<th>Cluster nodes</th>
<th>Role</th>
<th>HDFS</th>
<th>Spark</th>
<th>Cores</th>
<th>RAM(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td>reserved node</td>
<td>-</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 2</td>
<td>name node(active)</td>
<td>-</td>
<td>✓</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>Node 3</td>
<td>name node (standby)</td>
<td>-</td>
<td>✓</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>Node 4</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 5</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 6</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 7</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 8</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 9</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Node 10</td>
<td>worker</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5.1: Cluster nodes configuration and version information.

With FEBRL datasets are generated with comma-separated values (CSV) files with personal details. The columns in datasets are rec-id, given-name, surname, postcode, city, telephone-number, credit-card-number, income-normal, age-uniform, income, age, sex, blood-pressure.

5.1.2 Cluster environment

The experiments we perform, need Spark, which is an in-memory parallel computing engine. To provide a parallel computation platform, a multi-node cluster environment is needed. We used Hortonworks Data Platform (HDP) framework from the software vendor - Hortonworks. HDP is powered by the open source Apache Hadoop ecosystem used for storing, processing and analyzing in distributed environments. The cluster used for our evaluation is of 10 hosts (nodes) and the detailed information about each node is given in table Table 5.1. Each node is installed with centos 7 (x86_64) as the operating system. The following are the packages used in our work.

- **HDP**: As described, it is a comprehensive cluster managing framework. It can be seen as a single point of contact to do any configuration changes and serves as an umbrella of other packages in the cluster. The version used 2.6.4.0-91.

- **HDFS**: Hadoop Distributed File System (HDFS) is a distributed file system, developed to store the very large datasets reliably in a cluster-based environment (introduced in Section 2.4.1). The version used is 2.7.3.

- **Spark**: Spark is an open source, general purpose, in-memory, parallel, a highly scalable computational engine for massive scale data processing (introduced in Section 2.4.3). The version used is 2.2.0.

- **YARN**: Yet Another Resource Manager (YARN) is a layer of Hadoop, which does resource management and job monitoring/scheduling. It can be seen as data operating system, a global resource manager, and per-application manager for Hadoop [97]. The version used is 2.7.3.
• **Ambari**: Ambari is an open source management project to provision, manage, monitor and to secure Hadoop clusters. It is a part of HDP suite and version used is 0.1.0.

### 5.1.3 Submission of Spark Application to the Cluster

In our cluster environment, Sparks runs over YARN with HDFS as data layer (introduce in Section 2.4.3). In this section, we describe how the application is bundled and submitted to the cluster.

In our work, with the help of Maven\(^1\) build tool, an assembly jar (or “uber” jar) is created, which contains all the required dependencies. Once the application is bundled into a jar file then it can be launched with the `spark-submit` script. Below are the parameters set while submitting the `spark-submit` script to the cluster:

- **–class**: The classpath to the “main” class in the project.
- **–master**: The cluster manager, in our case it is “yarn”.
- **–deploy mode**: Since we want to run it on the cluster, the value is set as “cluster”.
- **–driver-memory**: Here, the memory, which should be allocated to driver process is to be given in “gb”. Since we are computing BDM, we have provided the value as 4gb.
- **–num-executors**: Here, we can specify the maximum number of executors that are allowed to be created. Spark may not create as many numbers of executors as given but it creates only, which are enough for the workload for the given configuration.
- **–executor-memory**: This value is used to allocate memory for each executor. This value is also varied according to the requirement of the experiment and is provided in the description of every experiment below.
- **–executor-cores**: It sets the number of cores for the given number of executors.
- **<application-jar>**: Path to the jar file of the application. In our case, it is an HDFS path: hdfs://ip/user/local user/jarfile.
- **[application-arguments]**: Arguments are provided.

These parameters will be varied according to the evaluation test case and we have provided these values before the commencement of every experiment.

\(^1\)https://maven.apache.org/
5.2 Experiment I: Finding optimal partitions and tuning the parameters

In this experiment, our goal is to find the effect of dataset partitions and other parameters for the Basic, BlockSplit, and PairRange approaches. Number of partitions of the input dataset has an effect on the overall execution and parallelization in the cluster environment. According to the Spark’s official documentation [98], the cluster is not fully utilized until each core is fully utilized i.e., the number of partitions should be as minimum as the number of available cores.

In experiment-I, spark-submit parameters are set as: num-executors=7 (considering workload), executor-memory =6gb, executor-cores=4, driver-memory=4gb, and test dataset used is DS1.

With 7 executors and 4 cores, gives to a total of 28 cores. According to the Sparks documentation, optimal performance is obtained at 2-3 tasks per CPU core in the cluster [98]. Hence, we have evaluated Basic (in Section 5.2.1), BlockSplit (in Section 5.2.2) and PairRange (in Section 5.2.3) approaches with varying partitions (multiples of 28) and the effect of match_tasks, ranges on BlockSplit, PairRange respectively.

Before going into results discussion, it is important to understand the anatomy of Spark jobs (as introduced in Section 2.4.3) for the three approaches. The breakdown of the three approaches into the respective number of “actions” and “stages” are shown in Figure 5.3. All the three approaches have an action saveAsTextFile, which takes the advantage of increasing partitions till a certain point. The optimal point is to be found out through the tests. The action sortByKey is present only in BlockSplit and PairRange. This action requires shuffling of data in order to sort, hence time for the execution of this action increases as partitions increase. The action collect increases proportionally with partitions, as the data from all the partitions need to be collected on to driver node, which is the part of the calculation of BDM.

5.2.1 Tuning of Basic Approach

To find out the optimal partition for Basic approach, tests are conducted with partitions as multiples of 28 i.e., 28, 56, 84... 280 and results are shown in Figure 5.2. We have found that Basic approach has the optimal partitions at 84 (i.e. 3×28). By this we can infer that action saveAsTextFile has its threshold at p=84.

5.2.2 Tuning of BlockSplit Approach

Here, we have tested BlockSplit approach with partitions p=28, 56 and 84 and varying the match_tasks m_t from 0 to 2000, to find out the effect of m_t. The results are shown in Figure 5.4.

From the Figure 5.4 and test results, we found that BlockSplit performed optimal at m_t=28 and m_t=1000. The execution times increased when partitions increased. This is because BlockSplit has the action sortByKey, which performs a shuffle for BDM calculation. And, as partitions increase this (shuffle) factor is dominating the
5. Evaluation

Figure 5.2: Execution times and partitions for the **Basic** approach.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Actions (jobs)</th>
<th>Wide range dependencies (stages)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td>saveAsTextFile</td>
<td>saveAsTextFile</td>
</tr>
<tr>
<td></td>
<td>first</td>
<td>mapToPair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapToPair</td>
</tr>
<tr>
<td><strong>BlockSplit</strong></td>
<td>saveAsTextFile</td>
<td>saveAsTextFile</td>
</tr>
<tr>
<td></td>
<td>sortByKey</td>
<td>flattenMapToPair</td>
</tr>
<tr>
<td></td>
<td>collect</td>
<td>sortByKey</td>
</tr>
<tr>
<td></td>
<td>first</td>
<td>collect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapToPair</td>
</tr>
<tr>
<td><strong>PairRange</strong></td>
<td>saveAsTextFile</td>
<td>saveAsTextFile</td>
</tr>
<tr>
<td></td>
<td>sortByKey</td>
<td>flattenMapToPair</td>
</tr>
<tr>
<td></td>
<td>collect</td>
<td>sortByKey</td>
</tr>
<tr>
<td></td>
<td>first</td>
<td>collect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mapToPair</td>
</tr>
</tbody>
</table>

Figure 5.3: The jobs and stages information of the three strategies.
5.2. Experiment I: Finding optimal partitions and tuning the parameters

Figure 5.4: The effect of match\_tasks (m\_t) on the execution times of the BlockSplit approach.

Advantage gained by the action saveAsTextFile. As the Basic approach does not have the action sortByKey, it has gained the advantage of increasing partitions up to 84.

**Influence of match\_task m\_t on BlockSplit:**

Figure 5.5 represents the time taken for each task in the BlockSplit approach with p=28 and with match\_tasks 50, 1000, and 2000. As we can infer that time taken for each task decreased as the value of m\_t increased. This is due to the fact that maximum size of the block is being reduced to 8×10^9, 4×10^8 and 2×10^8 for match\_tasks: 50, 1000, and 2000 respectively (calculated with Equation 4.2). Even though the size of max\_block size will decrease as match\_tasks increase, data shuffle also increases as match\_tasks increase. For DS1, shuffle read is 175MB, 539 MB and 670MB for m\_t=50, 1000, and 2000 respectively. Hence, the optimal value of m\_t will be dependent on the dataset and should be tuned. For the dataset DS1, we obtained at a optimal value of m\_t as 1000.

5.2.3 Tuning of PairRange Approach

In this, we find the optimal partitions for the PairRange and optimal ranges for DS1. Tests are conducted for three partitions namely 28, 56 and 84 with varying ranges, the resulting time can be seen in Figure 5.6. Even in PairRange optimal execution times are obtained for p=28, due to the same reasons discussed above for the BlockSplit approach.

**Influence of ranges \( \Gamma \) on PairRange:**

From the Figure 5.6, it is evident that for very low ranges (below 300), execution times are higher. This is due to, very small ranges give rise to a very large number of tasks
5. Evaluation

Figure 5.5: Task times for various match tasks of BlockSplit approach.

Figure 5.6: Execution times and ranges PariRange approach.
Figure 5.7: Task times and ranges of PariRange approach.

and this could compensate the advantage of small tasks with the communication overhead and task scheduling. Since we are running with 28 cores (7 executors × 4 cores), there exist 28 tasks for all the partitions above 28.

The individual execution times of 28 tasks for p=28, 56, and 84 are shown in Figure 5.7. We can infer that for large ranges are leading to higher execution times. Larger execution time is due to processing of larger blocks, as each range could be considered as a block. In the case of DS1:

- When Γ=50, each block size is 50 comparisons and total blocks of $8 \times 10^9$ should be processed.
- When Γ=10,000, each block size of 8000 comparisons and total blocks of $4 \times 10^8$ should be processed.
- When Γ=20,000, each block size of 2000 comparisons and total blocks of $2 \times 10^8$ should be processed.

For a very small value of Γ, larger blocks will be generated leading to bottlenecks. For the higher values of Γ, smaller blocks will be generated leading to communication overheads and increased percentage of shuffle reads. In our tests, we have got the threshold value of Γ at 800.

5.3 Experiment II: Robustness

In this experiment, we test the robustness of the two load-balancing approaches against the degree of skewness. This experiment is carried out with two types of datasets: Type I and Type II. In Type I datasets consist of one group of the duplicates and other groups of non-duplicates and in Type II datasets consists of varying proportions of duplicate groups of records.

*Spark-submit* parameters are set as: num-executors=7 (considering workload), executor-memory=6gb, executor-core=4, and driver-memory=4gb.
Configuration values use are: For Basic approach partitions, \( p=84 \), BlockSplit \( p=28 \) and match_tasks \( m_t=2000 \), and for PairRange \( p=28 \) and ranges \( \Gamma=600 \). Since, we are testing optimal performances, constant values of match_tasks (\( m_t \)) and ranges (\( \Gamma \)) are set for all the datasets used in this experiment. Next, we describe datasets considered and results of Type I (Section 5.3.1) and Type II (Section 5.3.2) respectively.

5.3.1 Type I

In Type I, we test the robustness by varying skewness of dataset, which is achieved by adjusting the blocking function. Let \( d \) be number the of duplicates and \( s \) be skewness, and \( e \) be the size of the dataset. Then, the number of duplicates \( d=s.e \), such that \( s \geq 0 \). To exemplify, let dataset be of 100 records and if \( s \) is 0 means, there are no duplicates. If \( s=0.3 \) means dataset contains 30 (30%) duplicates, which have the same blocking-key. Here, only one block exists in the whole dataset and other records will have unique blocking keys, thereby they do not form any other blocks. The dataset used is DS2 and its properties are shown in Figure 5.1.

![Figure 5.1: DS2 Dataset Properties](image)

<table>
<thead>
<tr>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
</tr>
<tr>
<td>Basic</td>
</tr>
<tr>
<td>BlockSplit</td>
</tr>
<tr>
<td>PairRange</td>
</tr>
</tbody>
</table>

![Figure 5.8: Execution times at different data skew percentages with DS2](image)

Results and Discussion:

The results are plotted for the above tests in Figure 5.8. At 0% skew, Basic approach performed better than the PairRange and BlockSplit, as there is no overhead of BDM
computations in Basic approach. As explained in Section 4.1, the Basic approach is not robust to skewed datasets, execution times almost increased linearly with skew percentages. Basic approach has an increase of 3595%, compared to this BlockSplit and PairRange have increased a maximum of 154% and 308% respectively. The increased execution times of PairRange compared to BlockSplit is due to the additional overhead of virtual indexing of comparison-pairs and index off-set. At 100% skew BlockSplit and PairRange were 92% and 87% better than Basic approach.

5.3.2 Type II

In Type II, the skew is defined as the percentage of records, which belong to one block compared to the total size of the dataset. In this fashion, the synthetic datasets are generated with varying skew from 0% to 100% by maintaining the number of blocks constant. For instance, 30% of skew of 100 records dataset means, the largest block contains 30 records and rest of the 70 records belong to 99 blocks. With this setup, DS3 with various skew factors are tested with three approaches and results are shown in Figure 5.9.

![Figure 5.9: Execution times and different data skews with DS2.](image)

<table>
<thead>
<tr>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>Basic</td>
</tr>
<tr>
<td>BlockSplit</td>
</tr>
<tr>
<td>PairRange</td>
</tr>
</tbody>
</table>

Results and Discussion:

At 0% skew situation, all the records are grouped into 100 blocks, hence there exist blocks of similar sizes. Therefore, Basic approach performed comparatively better
than other two. As there is an overhead calculation of BDM for BlockSplit and PairRange approaches.

As the skew factor increased, Basic approach execution time increased almost linearly up to 3478%. As expected, BlockSplit and PairRange managed to be in limits 203% and 250% respectively. The increase execution times of PairRange compared to BlockSplit is due to the same reasons discussed in Type-I. At 100% skew BlockSplit and PairRange were 91% and 88% better than Basic approach.

![Figure 5.10: Execution times and speedup values for all the three approaches using dataset DS4.](image)

### 5.4 Experiment III: Speed-up and Scalability

In this experiment, we test the utilization of nodes (scalability) in the cluster. Scalability is not only important factor in terms of speed of computation but also financial implications attached to it. In public cloud infrastructure models, vendors generally charge per used machine(node) and not on the percentage of the machine utilization. In this context, the utilization factor of each node becomes significant. To test the speed-up of Basic and the other two load-balancing strategies, the number of executors are varied from 1 to 8.

**Spark-submit parameters with DS4:** To achieve the range of executors from 1 to 8 the following parameters are set: executor-memory=1gb, executor-cores=3, and driver-memory=4gb, num-executors varied from 1 to 8.

In speed-up experiments, the following values are set: For Basic approach $p=84$, BlockSplit $p=28$, $m_t=2000$ and for PairRange $p=28$, $\Gamma=1000$. The values of match_tasks and ranges for DS4 are obtained from a series of tests shown in Figure 5.11. The resulting execution times and speed-up values of three approaches with DS4 are shown in Figure 5.10.
5.4. Experiment III: Speed-up and Scalability

Figure 5.11: Tuning of DS4 for BlockSplit and PairRange.

Results and Discussion:

At executor=1, execution is done almost (still running on 3 cores) in a sequential manner, at this point, the execution time of Basic approach is better than BlockSplit and PairRange approach. This is because Basic approach does not have the overhead of BDM calculation. Also, at executor=1, BlockSplit has better execution time than PairRange, this is because PairRange has an additional overhead of virtual indexing of comparisons and pair_index offset calculations apart from BDM calculations.

However, Basic did not scale after this and it has achieved a maximum speed-up of 1.3 till 8 executors. After 4 executors, it has become almost constant. Contrary to this both BlockSplit and PairRange approaches scaled almost linearly till 6 executors achieving scale-up of 4 and 3.5 respectively. As expected PairRange performed slightly better than BlockSplit as the executors increased because of the increased uniform distribution of the load. This is because increased executors compensated the additional overhead of PairRange over BlockSplit. At executors=8, BlockSplit and PairRange achieved 2.4 and 2.8 times higher speed-ups than Basic approach.
6. Conclusion and Future Work

6.1 Conclusion

In this work, for the pervasive problem of ER, we have analyzed and clearly illustrated the problem caused by data skew in ER, in the distributed environment. As over-sized blocks (due to data skew), were the reason for the bottlenecks in the whole execution process, we have adopted two accurate block distribution algorithms for our working environment, namely \texttt{BlockSplit} and \texttt{PairRange}. Both these algorithms work on the information of the blocks distribution statistics for the given dataset. \texttt{BlockSplit} algorithm will split the blocks, which are above a threshold value and the \texttt{PairRange} algorithm will distribute all the comparisons into equal (almost) ranges.

To evaluate the two algorithms adopted, a series of tests are conducted for: tuning the algorithms (performance), varying skew factors (robustness), and varying executors (speed-up). The datasets used are with records of order $10^4$, $10^6$, $3 \times 10^6$ with $5 \times 10^7$, $4 \times 10^{11}$, $1.8 \times 10^8$ comparisons respectively. All the respective results were discussed in detail.

Regarding the performance, after tuning for the dataset DS1, both \texttt{BlockSplit} and \texttt{PairRange} approaches achieved an improved execution times by 23% and 15% compared to \texttt{Basic} approach respectively.

Regarding the robustness, both the adopted algorithms are proved to be highly robust, even as the percentage of skew increased the percentage of increase comparing to \texttt{Basic} approach was minimal. At 100% skew (for both DS3 and DS4), \texttt{BlockSplit} and \texttt{PairRange} achieved more than 90% and 88% of improved execution times compared to \texttt{Basic} approach.

Regarding the speed-up, \texttt{BlockSplit} and \texttt{PairRange} achieved 2.4 and 2.8 times higher speed-up compared to the \texttt{Basic} approach with DS4.

6.2 Future Work

So far in this work, we have taken up the problem of data skew in the ER computations, especially in the context of the distributed environment. In this limited time,
6. Conclusion and Future Work

we have studied accurate block distribution algorithms, to be specific only two. Due to time limitations, we haven’t had the opportunity to explore more options, apart from the discussed work so far. Here, we take the opportunity to write down few directions of possible extension of this work.

**More Algorithms:** As also discussed in Chapter 4, there are also few more accurate block distribution algorithms: MaxBlock, BlockSlicer, and MSBlockSlicer. To the best of our knowledge, there is no work has been done regarding these with Spark.

**Scalable Load-balancing Strategies:** The major limitation of accurate block distribution based algorithms is that full details are to be fetched to driver node. This will face scalability problems when dealing with billions of records. Even in our work, 1GB of allocated memory was not sufficient for datasets with $10^7$ records and we faced “out of memory exceptions” (we increased it to 4GB to continue our tests). One direction of work is to develop strategies based on sampling rather than full details. In [94], datasets are profiled based on the sketch is proposed to address the problem discussed above.

**Changes to Spark:** The problem of load-imbalance arises due to the unequal distribution of data across the nodes. In our work, we tried to deal with this explicitly outside Spark. One direction of work could be, making Spark aware of the dataset profile. There, by Spark could automatically respond by dynamic-repartitioning in accordance with skew. In [68], a similar work is proposed.
Bibliography

on Page iii, 1, and 31)

S. Shenker, and I. Stoica, “Fast and interactive analytics over hadoop data with


resolution,” in Data Engineering (ICDE), 2012 IEEE 28th International Con-
ference on, pp. 618–629, IEEE, 2012. (cited on Page ix, xi, 1, 29, 37, 38, 39, 44,
and 45)

www.smartprix.com/mobiles/apple-iphone-8-p1101r03th5u; visited on March
10th, 2018. (cited on Page xi and 8)

(cited on Page xi, 1, 8, 9, 11, 15, 16, 17, and 18)


https://spark.apache.org/docs/latest/cluster-overview.html; (cited on Page xi
and 25)

https://intellipaat.com/tutorial/spark-tutorial/spark-architecture/; (cited on
Page xi and 26)


p. 28, 2008. (cited on Page 1)


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

Magdeburg, den April 10, 2018