UNIVERSITY OF MACEDONIA
GRADUATE PROGRAM
DEPARTMENT OF APPLIED INFORMATICS

PARALLELIZING ENTITY RESOLUTION METHODS FOR BIG DATA

Master Thesis

By

Vasileios Tsogkas

Thessaloniki, February 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΑΡΑΛΛΗΛΟΠΟΙΗΣΗ ΜΕΘΟΔΩΝ ΔΙΕΥΘΕΤΗΣΗΣ ΟΝΤΟΤΗΤΩΝ ΓΙΑ ΜΕΓΑΛΑ ΔΕΔΟΜΕΝΑ

Διπλωματική Εργασία

του

Βασίλειου Τσόγκα

Θεσσαλονίκη, Φεβρουάριος 2022

Abstract

Entity Resolution (ER) is the process of locating records which represent the same real-world entity, within a single dataset or across different datasets. ER exists for several years now and has been evolving constantly, since it has to keep up the pace with the developments in technology, as well as in the field of data management. All these years, various techniques have been used for the implementation of the ER process, like blocking, filtering, and matching, in order to improve its performance and effectiveness. However, ER faces new challenges in the age of big data analytics we live in, since traditional methods of handling data have not proved very efficient. Hence, ER in turn must evolve further, so as to adapt to the modern world of Big Data analytics. In this work we study the ER process, how it is divided in stages and present popular methods used in each stage. We focus on Blocking techniques and specifically on Improved Suffix Array Blocking with Bloom Filters. After implementing this method serially, we study how to apply parallelization, using Apache Spark. We conduct comparative experiments between the serial and parallel execution, present the results and examine the significant improvement in efficiency, when the process is executed in parallel. Our conclusions indicate that ER methods, if applied in a distributed manner, are capable of handling Big Data.

Keywords: Entity Resolution, Big Data, Blocking, Filtering, Inverted Index, Suffix Array Blocking, Bloom Filter, parallel execution, Apache Spark, Scala, Java

Περίληψη

Η *Διευθέτηση Οντοτήτων* (*Entity Resolution*, ER) είναι μία διαδικασία με την οποία εντοπίζονται εγγραφές, οι οποίες αναφέρονται στην ίδια οντότητα του πραγματικό κόσμου, εντός ενός συνόλου ή μεταξύ περισσοτέρων συνόλων δεδομένων. Η Διευθέτηση Οντοτήτων υπάρχει εδώ και χρόνια και εξελίσσεται συνεχώς, αφού πρέπει να συμβαδίζει με τις εξελίξεις τόσο στον χώρο της τεχνολογίας όσο και στην διαχείριση δεδομένων. Σε όλο αυτό το διάστημα, για την υλοποίηση της συγκεκριμένης διαδικασίας και την βελτίωση της απόδοσης και της αποτελεσματικότητάς της, έχουν χρησιμοποιηθεί διάφορες τεχνικές, όπως ομαδοποίηση (blocking), φιλτράρισμα (filtering) ή αντιστοίχιση (matching). Ωστόσο, η ER αντιμετωπίζει νέες προκλήσεις τα τελευταία χρόνια, αφού περισσότερο παραδοσιακές μέθοδοι, δεν είναι και τόσο αποδοτικές ως προς την επεξεργασία μεγάλων δεομένων (Big Data). Συνεπώς, και η Διευθέτηση Οντοτήτων με τη σειρά της πρέπει να εξελιχθεί περεταίρω, ώστε να προσαρμοστεί στον σύγχρονο κόσμο της ανάλυσης μεγάλων δεδομένων. Σε αυτή την εργασία, μελετούμε την διαδικασία Διευθέτησης Οντοτήτων, τα στάδια στα οποία διαρθρώνεται και παρουσιάζουμε κάποιες δημοφιλείς μεθόδους που χρησιμοποιούνται σε κάθε στάδιο. Εστιάζουμε σε τεχνικές ομαδοποίησης και ειδικά στην τεχνική Suffix Array Blocking with Bloom Filter. Αφού υλοποιήσουμε την συγκεκριμένη τεχνική σειριακά, μελετούμε πώς θα εφαρμόσουμε παραλληλοποίηση με την χρήση του Apache Spark. Διεξάγουμε συγκριτικά πειράματα μεταξύ της σειριακής και της παράλληλης εκτέλεσης της διαδικασίας, παρουσιάζουμε τα αποτελέσματα και εξετάζουμε την σημαντική βελτίωση στην απόδοση, όταν η διαδικασία εκτελείται παράλληλα. Τα συμπεράσματά μας υποδεικνύουν ότι οι μέθοδοι που χρησιμοποιούνται στην Διευθέτηση Οντοτήτων, εάν εφαρμοστούν με έναν κατανεμημένο τρόπο, είναι ικανές να διαχειριστούν μεγάλα δεδομένα.

Λέξεις Κλειδιά: Διευθέτηση Οντοτήτων, μεγάλα δεδομένα, ομαδοποίηση, φιλτράρισμα, παράλληλη επεξεργασία

Acknowledgements

I would like to express my sincere gratitude to the supervisor of this master thesis, Assistant Professor in the Department of Applied Informatics, Georgia Koloniari, for her constant support and guidance throughout the completion of this study.

I would also like to thank the members of the Committee, Associate Professor in the Department of Applied Informatics Theodoros Kaskalis and Assistant Professor in the Department of Applied Informatics Papadimitriou Panagiotis, for their contribution to the study.

Finally, I would like to express my deepest gratitude to my wife Fevronia, for her love and patience, and to my beloved children Sofia, Spiridon and George Prodromos, who really embraced my effort and supported me in every way.

Table of Contents

Abstract i
Περίληψη ii
Acknowledgements iii
Table of Contents iv
TABLES vi
FIGURES vii
1. Introduction 1
1.1 The Problem 1
1.2 Goal of this thesis 1
1.3 Outline of this thesis 2
2. Theoretical Background 3
2.1 Techniques 3
2.2 Matching 4
2.3 Similarity Functions 6
2.4 Complexity 7
2.5 Blocking 8
2.6 Filtering 11
3. Selected Method 12
3.1 Suffix Array Blocking 12
3.2 Improved Suffix Array Blocking 15
3.3 Bloom Filters 17
4. Implementation Issues 19
4.1 Apache Spark 19
4.1.1 What is Apache Spark 19
4.1.2 How it works 21
4.1.3 Spark's Distributed Architecture 24
4.1.4 Spark RDDs, DataFrames and Datasets 25
4.2 Scala 26
4.3 Implementations 27
4.4 Serial Implementation 28
4.5 Parallel Implementation 29
5. Experiments 31
5.1 Serial Execution vs Parallel Execution 32

5.1.1 Stage 1 33
5.1.2 Stage 2 34
5.1.3 Stage 3 35
5.1.4 Overall Process 36
5.2 Resources Usage 37
6. Conclusions and Future Work 40
6.1 Conclusions 40
6.2 Limitations of this study 40
6.3 Future Work 41
7. Bibliography 42

TABLES

Table I: Generated suffixes of minimum length $lmin$ = 4, from “JohnThompson” BKV..... 12
Table II: Inverted Index structure, sorted alphabetically 13
Table III: Improved Suffix Array Algorithm 16

FIGURES

Figure 1: Naive brute force approach 7
Figure 2: Sorted Neighborhood Blocking, with Sliding Window 10
Figure 3: Sorted Blocks with Sliding Window 10
Figure 4: Bloom Filter implementation using k hash functions (Vries, et al., 2011) 18
Figure 5: Bloom Filter implementation using one hash function 18
Figure 6: Spark's example of a Directed Acyclic Graph (DAG) 21
Figure 7: Spark Driver creating Jobs, Jobs creating Stages and Stages creating Tasks 23
Figure 8: Spark's distributed architecture (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020) 24
Figure 9: Overall Improved Suffix Array Blocking approach 28
Figure 10: Stage 1 - Comparative performance graph 33
Figure 11: Stage 2 - Comparative performance graph 34
Figure 12: Stage 3 - Comparative performance graph 35
Figure 13: Overall Process comparative performance graph 36
Figure 14: Serial Execution - CPU Utilization 37
Figure 15: Serial Execution – PostgreSQL CPU and Memory Usage 37
Figure 16: Parallel Execution – CPU Utilization 38
Figure 17: Parallel Execution – CPU and Memory Usage 38
Figure 18: Serial Execution – Active Database Sessions 39
Figure 19: Serial Execution – Number of transactions per second 39
Figure 20: Parallel Execution – Active Database Sessions 39
Figure 21: Parallel Execution – Number of transactions per second 39

1. Introduction

1.1 The Problem

For several years now, the era that followed the rapid evolution of computer systems is characterized by the development of all kinds of applications and services, aiming at improving one's everyday life, in terms of work or personal life (Enterprise Applications, Cloud Computing, Email, Social Networking, newsfeed, Streaming Services for music or movies etc.). The global expansion of such services demands the support of huge infrastructures, which end up bearing countless data volumes, stored, and accessed in many different methods. Nowadays, the world has reached the point where a significant part of one's social and professional activity is carried out through mobile applications or services. More particularly, anyone can register in several platforms using more than one personal or professional accounts. This leads to a situation where an individual is referred to by a number of separate applications in various ways (Bhattacharya & Getoor, 2007). And there is more to it, since the word individual is not necessarily confined only to humans but may also represent any other object which constitutes an entity inside an application (organizations, products, brands, services and so on).

So, in several cases, we come upon datasets where multiple records, and not necessarily identical, may refer to the same real-world entity (Benjelloun, et al., 2009). This happens usually when a dataset does not use identifiers for the objects it stores (Bhattacharya & Getoor, 2007). It also happens when a database originates from the consolidation of smaller ones, since a real-world entity may be described in various ways across multiple datasets (Vassilis Christophides, et al., 2015). The problem we have to deal with is the fact that referencing the same real-world entity in numerous and perhaps distinctive ways is an indication of information redundancy across different systems or the existence of duplicate records, in case of a single dataset.

In order to solve this problem, we use *Entity Resolution* ("ER"), which is the process of identifying (and sometimes merging) distinct and unrelated descriptors which allegedly represent the same real-world object. Locating such records, especially when no unique identifiers have been used in a dataset, is not a simple process, and it requires a bit of guesswork (Benjelloun, et al., 2009). Ironically, ER itself is known by several names, such as *record linkage*, *reference reconciliation* or *deduplication*, and although it has a long history (since 1950s) it still remains active, especially because of the evolution of "Big Data" (Getoor & Machanavajjhala, 2012).

1.2 Goal of this thesis

Discovering similar records inside datasets has been proved to be a substantial procedure when it comes to organizations that handle huge data volumes. For example, locating duplicate records inside a database or consolidating a company's customer databases stemming from several subsidiaries. After all, applications have been storing and managing a large amount of data (structured or unstructured) for years, therefore an application's

efficiency and performance highly depends on the quality and quantity of its data. Since ER aims at locating similar records inside a dataset and solves the problem by merging or deleting them, it can and probably has already become a powerful tool towards data volume reduction and an increase in data quality.

The goal of this thesis is to prove that if we use parallelization techniques in the implementation of ER methods, not only can we achieve a significant improvement in the performance of the process, but also this performance boost comes without any tradeoff regarding the quality of the results. Specifically, we study blocking techniques and implement Improved Suffix Array Blocking with Bloom Filters method (Vries, et al., 2011), both serially and in parallel. Regarding the parallel implementation we use Apache Spark, which performs exceptionally fast and the results of experimenting on several datasets of various sizes, shows that ER process shows a remarkable performance on very large datasets. Therefore, if executed in parallel ER is able to process big data.

1.3 Outline of this thesis

The introductory part of this study presents the circumstances that led to the development of ER process, its purpose, and the goal of this thesis.

Chapter 2 refers to the theoretical background of ER and presents techniques, used to implement the stages of the process. The chapter focuses on matching, blocking, and filtering techniques and presents the most popular of each.

In Chapter 3, Suffix Array Blocking is presented in detail, as well as its improved version, which is the one we focus in this work. Bloom Filters are also explained.

Chapter 4 contains details of the implementation of Improved Suffix Array Blocking with Bloom Filters. Apache Spark and the Scala programming language, which are the tools that are used for the parallel implementation, are presented, also. We explain how this method is implemented, both serially and in parallel.

Chapter 5 presents the results of the experiments conducted, regarding the performance and the way each method utilizes the available resources.

In Chapter 6, we present our conclusions on how much performance gain there is, by using Apache Spark to parallelize ER methods. Some limitations, regarding the experiments, are mentioned, as well as some interesting matters for future work.

2. Theoretical Background

As its name implies, ER's scope are entities of the real world. From this point of view, our attention is drawn to those datasets, where each record represents a real-world entity. An entity, inside a dataset, is described by the relative record's set of attributes, which are usually in the form of <key, value> pairs. Of course, it depends on the storage method each application uses. Instead of <key, value> pairs, data may be stored in a relational database (where the equivalent of the key is the column name) or in any other structured or unstructured form. However, lately, in cases where a large amount of data needs to be processed, in contrast with more traditional and concrete data models, like the relational, the world of data has turned to more robust and flexible models such as the attribute-value sets (Vassilis Christophides, et al., 2015). Intuitively, we could assume that a record's attributes, combined together, compose its distinct identifier. From now on we refer to such an identifier, as *entity*. For example, regarding a dataset where records contain personal information about people, an entity could have the form:

e = {id: 0001, lastame: xxxxxxxx, firstname: yyyyyyy, gender: M, …}

Similarly, we assume that a dataset of records that corresponds to entities, constitutes a collection of entities.

In relation to our problem, sometimes we come upon the situation where, within a collection of entities (or across different collections), two distinct entities refer to the same real-world object. In this case, these entities are considered duplicates and some action must be taken. ER applies in locating and eliminating duplicate entities:

- within a single entity collection, a case known as deduplication
- across multiple entity collections, a case known as record linkage (Papadakis, et al., 2019)

2.1 Techniques

The ER process has evolved over the years and the wider the spectrum of applications that make use of it becomes, the more techniques are employed towards producing more efficient and accurate results. These techniques are grouped depending on the stage they are used at. In its basic approach, ER compares entities in order to locate the duplicate ones. This is the *matching stage*, and it is carried out by applying a proper similarity algorithm on each pair of entities. With respect to the range of the entities taken into account, so as to clarify whether two entities are matching or not, we come upon two approaches:

- Pairwise ER, where the decision to match a pair of entities considers only the two entities compared, ignoring any common relations to other entities whose information could affect the result of the comparison (Benjelloun, et al., 2009), (Getoor & Machanavajjhala, 2012).
- Collective ER, on the other hand, apart from the two entities under comparison, makes use of information extracted from entities related to the above. Basically,

collective ER goes beyond the limitation of the strict similarity measures by exploiting the relations between entities and gathering additional information. This way, the relational evidence may lead to a possible match in cases where a strict similarity function would fail due to linguistic divergencies (Bhattacharya & Getoor, 2007), (Vassilis Christophides, et al., 2015).

In this work, we deal with pairwise ER.

The simplest and more obvious method of locating duplicates in a collection of entities (or across more collections) is to compare each entity with all others, in pairs. However, taking into consideration the huge size of modern datasets, this brute force approach seems infeasible due to an excessive number of comparisons, so other techniques come into consideration. To be able to apply an optimization, first we need to figure out which part of the brute force solution can be left out of the process. Actually, there is no point in comparing clearly dissimilar entities (e.g., men to women), so we must find a way of eliminating pointless comparisons. An effective and quite popular technique, ideal for our purpose, is the *blocking* technique, which is incorporated as a new stage, before matching, within the ER process. The idea of blocking is to group the entities in smaller collections in such a way, that each collection contains only entities which have a strong possibility of matching. This way no unnecessary comparisons occur.

Lately, ER process, even if it includes a blocking stage, has also been considered as open to further improvement, thereby the *filtering stage* has been introduced. Filtering is a part of the matching process; its scope is a specific group of entities (produced in the blocking stage) and it precedes the matching stage. In general, during the filtering stage, an entity, within its block, is assigned with a subset of matching candidates, filtering out entities that cannot match with it.

2.2 Matching

As we have already mentioned, records inside a dataset are defined by a set of attributes, which compose an identifier called entity. ER aims at locating entities that refer to the same real-world object, therefore it is safe to assume that values of an entity's attributes, represent details about the corresponding real-world object. For example, regarding a person, some of the attributes might be one's personal information (firstname, lastname, date of birth etc.). Thus, it would make sense to consider as matching records, those which share the same values in a significant set of attributes, and this set of attributes must guarantee as much as possible the uniqueness for each entity. We should point out, though, the fact that not all attributes need to be compared between entities, because depending on each dataset's scope, highly mutable information may be stored as well (e.g., work address or product's order information). Usually, these comparisons are implemented inside a matching function, which is adapted to the characteristics of the dataset under examination.

In general, the matching function compares two entities via a carefully selected subset of their attributes and decides whether they are matching or not. An ideal matching function would locate all the matching entities inside a collection, but this is not realistic, since entities that

refer to the same real-world entity, due to several reasons (such as different input practices or synonyms usage), can be slightly or considerably different. Taking this into account, we can comprehend the fact that, a matching function is actually able to detect only a fraction of the existing matching entities. On the other hand, the limited number of attributes used for identification, could result in a number of false matches, a case known as *false* positives, because distinct entities may share the same attributes (e.g., it is common for relatives to be namesakes of each other).

As we have already mentioned, an entity is essentially a dataset record, which represents a real-world object and is defined by a set of attributes in the form of <key, value> pairs. Furthermore, a convenient subset of these attributes constitutes a unique identifier for the entity. Expressing the above formally, let A be a set of attribute names and V be a set of attribute values. We also consider each dataset as a collection of entities, denoted as $\mathcal{E}$. Then an entity $e_i \in \mathcal{E}$ can be defined as $e_i = \{(a_i, v_i) | a_i \in A, v_i \in V\}$ and two entities $e_i, e_j \in \mathcal{E}$ match if they represent the same real-world object (Vassilis Christophides, et al., 2015).

ER's assignment is the detection of matching entities within a collection of entities or across two or more collections (Papadakis, et al., 2019). The techniques used to accomplish this are based on the definition of a boolean matching function *M* which reviews two entities $e_i, e_j \in \mathcal{E}$ and decides if they refer to the same real-world entity or not. Therefore, when $M(e_i, e_j) = true$, then e_i and e_j match, denoted by $e_i \equiv e_j$. On the contrary, if $M(e_i, e_j) = false$, the matching function considers that $e_i \; and \; e_j$refer to different entities, , denoted by $e_i \not\equiv e_j$ (Papadakis, et al., 2019), (Vassilis Christophides, et al., 2015).

Intuitively, we can assume that two entities can be considered as matching when there is enough similarity between their equivalent attribute values (Vries, et al., 2011). Hence, the matching function is essentially a wrapper to the actual entity comparison tool, a similarity function *sim*, which calculates at what degree two entities are similar to each other. Similarity functions, generally, consider various criteria to perform the desired comparisons between two entities. The final decision depends on a threshold value θ. When the similarity between two entity profiles e_i, e_j exceeds the threshold θ, then $M(e_i, e_j) = true$, otherwise $M(e_i, e_j) = false$. More formally:

$$M(e_i, e_j) = \begin{cases} true, if \; sim(e_i, e_j) \geq \theta \\ false, otherwise \end{cases} \quad , \quad where \; e_i, e_j \in \mathcal{E}$$

(Vassilis Christophides, et al., 2015)

The fact that ER covers a large spectrum of applications, many of which use data from various sources, or the data are characterized by significant heterogeneity, is the main reason why a strict matching function cannot do the job. That is why the matching function should be able to detect similar and not strictly identical entity profiles, exploiting the fact that a high value of similarity between t (Левенштейн & Levenshtein, 1965)wo entities implies that they are likely to match.

(Vassilis Christophides, et al., 2015)

Another significant matter is the effectiveness of the matching function. It could prove to be quite helpful in terms of finding the most suitable function for an examined dataset. A matching function's effectiveness can, in fact, be measured. Several metrics have been introduced, but we mention only a couple of them. Let *P* be the set of all the actual matching

entity pairs and *F* be the set of the pairs that our matching function detected. Then we can define the following metrics:

- *Recall*: $\frac{|P \cap F|}{|P|}$, the fraction of the matches identified
- *Precision*: $\frac{|P \cap F|}{|F|}$, The fraction of the matches identified, (Papadakis, et al., 2019)

An ideal matching function would detect all existing matches, and nothing more, hence both the above quotients' result would be 1 (100%). An effective matching function, on the other hand, tends to maximize the identified existing matches, while minimizing the false matches.

2.3 Similarity Functions

A similarity function is a real-valued function, and its purpose is to measure how similar two objects are, in a quantitative manner. Such functions produce a result within a specific range. Usually, the boundaries of this range correspond to the function's result when two objects are identical and the result when two objects are dissimilar, respectively. There exist various similarity functions, but in this work, we are interested specifically in those which compare text objects. Such functions may be character based or token based and some of them are presented, briefly, as follows.

Character based similarity functions, in general, consider the edit operations (Insertion, Deletion, Substitution) required, so that they can transform one string into another:

- **Levenshtein distance** (Левенштейн & Levenshtein, 1965) (Jaro, 1989)
  This metric calculates the minimum number of edit operations required to transform the one string into the other. Edit operations are Insertion, Deletion or Substitution of a single character and Transposition of two adjacent characters.

- **Jaro Similarity** (Jaro, 1989)
  Let s_1, s_2 be two compared strings of length $|s_1|$ and $|s_2|$, respectively, and $|c|$ the number of common characters between $s_1 and\ s_2$. Two characters are considered as common if they are the same and their positions within the compared strings differ no more than $\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor - 1$. Let also *t* be the number of transpositions which must occur, between matching characters being in different positions. Then
  $$Jaro(s_1, s_2) = \frac{1}{3}\left(\frac{c}{|s_1|} + \frac{c}{|s_{2|}} + \frac{c-t}{c}\right)$$

Token based similarity functions inspect the set of tokens, which exist within two strings (e.g. n-length substrings). Let S_1, S_2 be two sets of tokens, derived from each of the two strings under comparison:

- **Jaccard Similarity**
  This metric compares the number of common tokens, divided by the number of all unique tokens.
  $$Jaccard(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

- **Dice Similarity**
  This metric compares the number of common tokens, divided by the average number of tokens per string.

$$dice(S_1, S_2) = \frac{2\,|S_1 \cap S_2|}{|S_1| + |S_2|}$$

- **Overlap Similarity**
  This metric compares the number of common tokens, divided by the number of unique tokens in the smaller set.

$$overlap(S_1, S_2) = \frac{|S_1 \cap S_2|}{\min\left(|S_1|, |S_2|\right)}$$

2.4 Complexity

The naive approach of ER is the brute force approach, where each entity must be compared to all the others (Figure 1), while searching for possible matches. This results in a quadratic time complexity process (Papadakis, et al., 2019), a problem which can become even worse with respect to the size of the examined dataset. Not to mention the fact that the matching function, itself, is likely to be computationally expensive.

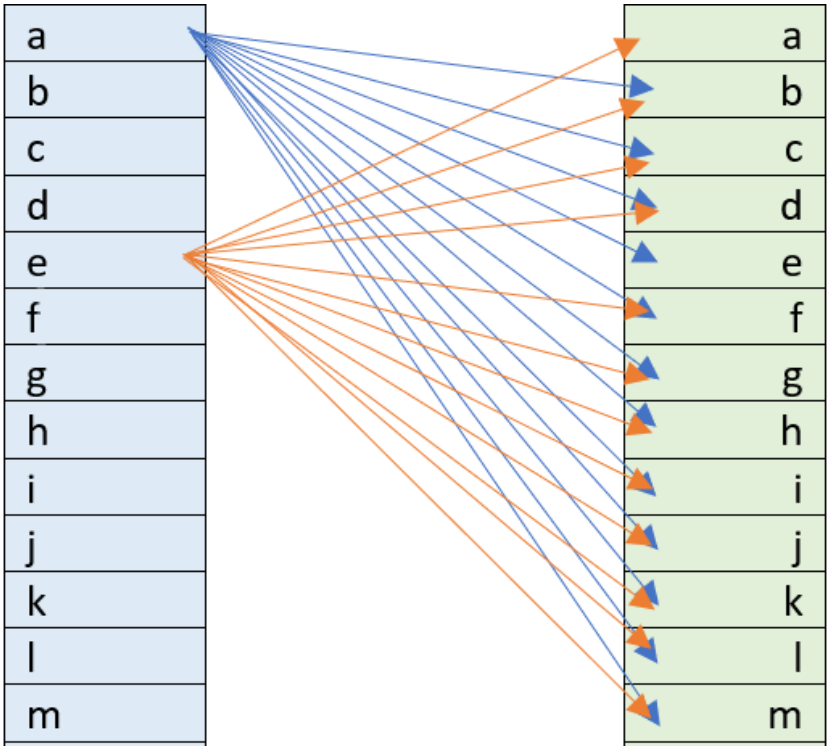


Figure 1: Naive brute force approach

Whatever optimization method we decide to apply, the goal cannot be simply to cut down the number of comparisons, and therefore reduce time complexity. Caution needs to be taken in order to avoid missing a significant number of potential matches. For this reason, we have to be able to evaluate ER techniques in terms of performance, using the following quantitative evaluation criteria:

- *Effectiveness*: the amount of the actual matches identified
- *Efficiency*: the computational cost for the identification of the matches (could be measured by the number of comparisons that took place during the ER process)

Considering the fact that the brute force approach maximizes the effectiveness, since it can locate all actual matches, but also diminishes efficiency due to the considerably high computational cost leads us to the estimation that there is a tradeoff between Effectiveness and Efficiency. Intuitively, as far as ER is concerned, to improve efficiency, we must sacrifice effectiveness, to some extent. In fact, our realistic purpose is to optimize efficiency at the lower possible cost of effectiveness.

Consequently, various techniques have been adopted with a view to reduce the computational cost of the ER process and optimize its performance, while at the same time trying to minimize any loss in precision, so as to make ER applicable to real world (large) datasets. The two most popular techniques, and the ones this work examines, are Blocking and Filtering.

2.5 Blocking

The main reason behind the quadratic complexity of a brute force pairwise ER approach is the vast number of comparisons, which take place among entities. Hence, the primary goal is the reduction in comparisons. Of course, the perfect solution would be to examine only the actual matching entities, but this is not feasible. After all, if we were able to know with certainty which entity pairs match no action would have to be taken. However, we must be careful in keeping the matches ratio as high as possible to maintain the credibility of ER process. Considering the above, it is obvious that the basic idea of blocking is to improve ER efficiency by diminishing as many as possible of the estimated comparisons, while at the same time keep the anticipated missed matches at the lowest possible level, so that ER effectiveness is not adversely affected to a large extend (Papadakis, et al., 2019).

Blocking, basically, distributes entities into separate blocks, based on some preliminary similarity criteria. Entities that meet the same similarity criteria are considered as possible matches and they are placed within the same blocks. An entity may be placed in more than one blocks, depending on whether the applied technique permits it or not. After the distribution phase is completed, each block contains entities that are likely to be similar. The most important part, though, is the fact that, entities which have a very low probability of being a match, do not co-exist within the same blocks. So, when the matching phase is reached, which is the next step in the ER process, comparisons are performed solely between entities within the same block (while rather pointless comparisons are avoided).

It is a fact that no ideal blocking algorithm exists, mainly because the data themselves are not likely to be ideally registered inside datasets. Heterogeneous sources, misspellings, different

input methods, etc. result in further divergence, concerning the similarity criteria. Hence, it is reasonable to assume that this technique comes to some cost. There is the possibility of missing some of the existing matches due to the fact that the blocking algorithm might place two actually similar entities in different blocks. Thus, those two objects are never compared to each other, a case known as *false negative*. As a rule, a blocking algorithm is considered successful if it manages to maintain a decent balance between effectiveness and efficiency. (Papadakis, et al., 2019)

Up to this day, various blocking methods have been implemented, some of which are the following:

- **Standard Blocking (or Traditional Blocking)** (Sunter & Fellegi,)
  A carefully selected entity attribute (or set of attributes) serves as the blocking key. Entities that share the exact same blocking key are placed within the same block. This method is exceptionally accurate considering the potential matches within a block, due to the strict blocking key policy. However, the slightest misspelling in one of the blocking key attributes, leads to a missing match.

- **Suffix Array Blocking** (Aizawa & Oyama, 2005)
  A carefully selected set of attributes, combined together, constitutes an identifier for each entity, called a Blocking Key Value (BKV) (Vries, et al., 2011). From each BKV, all possible suffixes (usually longer than a minimum length value) are generated. Each distinct suffix servers as a blocking key and all entities with common suffixes deriving from them, are placed in the same block. It is obvious that each entity is placed in as many blocks as there are suffixes generated by its BKV. This method is more effective than Standard Blocking because it is not affected by misspellings in the largest (starting) part of the BKV. However, misspellings towards the end of the BKV (especially near the minimum length value character) lead to missed matches.

- **Q-Grams Blocking** (Christen, 2012) (Papadakis, et al., 2015)
  In this method, BKVs are constructed according to Suffix Array Blocking, but now, each BKV generates all possible q-length substrings (e.g., for *q = 4,* BKV *"JohnDoe"* generates 4-grams *{John, ohnD, hnDo, nDoe }*). Each distinct q-gram serves as a blocking key and all entities sharing a q-gram are placed in the same block. This method is more resilient that Suffix Array Blocking because it is not affected by any misspellings anywhere across the BKV. It is, however, less efficient due to additional redundancy and because block sizes are not limited (Papadakis, et al., 2019).

- **Sorted Neighborhood Blocking** (Mauricio A. Hernández, 1995)
  As in Standard Blocking, a proper set of attributes is selected to form the blocking key. All blocking keys are sorted alphabetically, and the sorting is extended to the referenced entities. Using a sliding window, of fixed size *x*, on the sorted entity list, all the entities within the widow, are compared to the last one in it **(**Figure 2**)**.

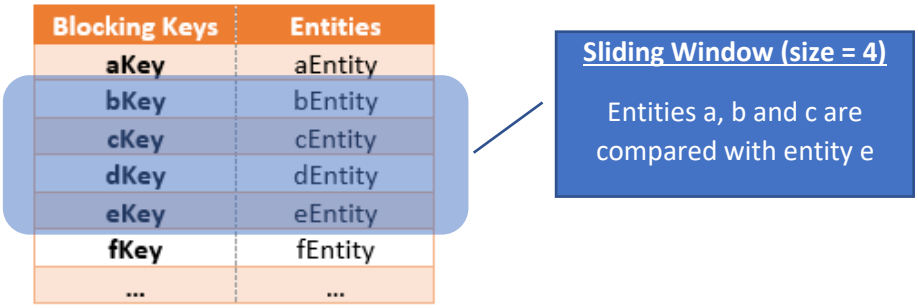


Figure 2: Sorted Neighborhood Blocking, with Sliding Window

- **Sorted Blocks** (Draisbach & Naumann, 2011)
  Similarly to Sorted Neighborhood, a proper set of attributes is selected to form the blocking key, and blocking keys and their referenced entities are sorted. Entities are distributed in blocks according to some prefix of the blocking keys. The comparisons take place in two stages. During the first stage, all entities within a block are compared to each other. At the second stage, a sliding window of fixed size begins by including the last *k* entities of the current block and the first entity of the next block. Each time the window slides by one position, towards the next block, all comparisons between entities of different blocks are performed. The window stops sliding when it reaches the first *k* entities of the next block **(**Figure 3**)**.

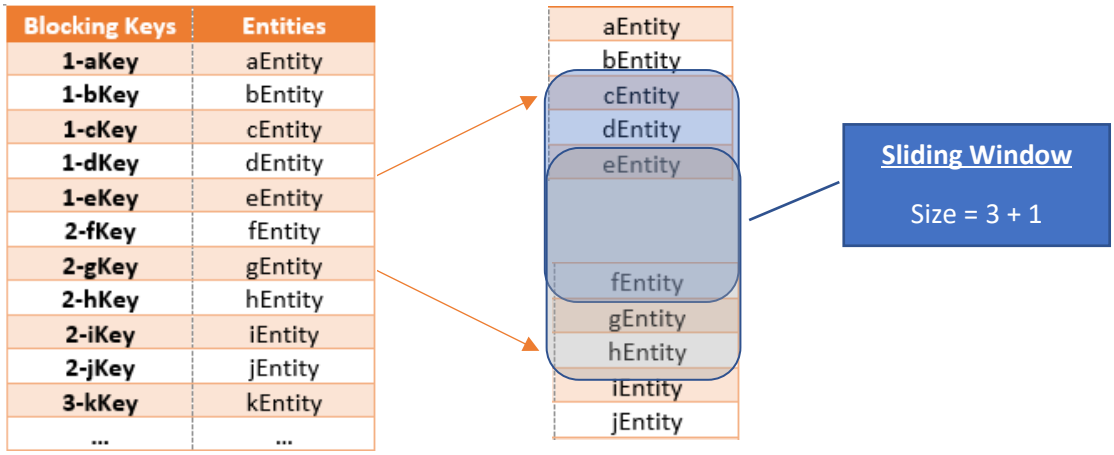


Figure 3: Sorted Blocks with Sliding Window

- **Token Blocking** (Papadakis, et al., 2011)
  In this method, all possible tokens must be generated from all attribute values of each entity. Every distinct token serves as a blocking key and all entities referenced by a blocking key are placed within the same block. It is of no significance whether common tokens between different entities stem from different attributes or not.

2.6 Filtering

Blocking generates several blocks based on some preliminary similarity criteria (e.g., *n-length* prefixes, for texts). Ideally, entities within a collection would be evenly distributed within blocks, but this is practically infeasible. If, for example, the blocking key is a 4-length prefix, the size of the derived blocks vary depending on the popularity of the prefix ("inte-" prefix is quite common in words, and probably produces a large block, "brut-", on the other hand, is not that common, thus a small block arises).

Since blocking techniques aim at improving ER efficiency, the fact that we may end up having to process blocks with significant differences in size, indicates that block processing times vary, respectively. So, there is a possibility that the improvement we expected is not achieved, in the end. One way to overcome such a setback, is to enforce some constraints, like maximum block size. While this solution secures the efficiency of ER process at some extent, there are still a few drawbacks (e.g., what happens with entities that satisfy the same blocking criteria, but are placed inside different blocks? How much more complex may the blocking algorithm become?).

A different approach to improve blocking efficiency is to include *filtering*, as a new stage in the ER process. Filtering takes place after the blocking process and before the matching phase. That is why, sometimes it is referred as post-blocking or meta-blocking, and some other times as initial matching process. Filtering, essentially, performs quick and not that accurate comparisons between entities within a block, with the purpose of excluding from the subsequent matching phase entities that are not likely to be a match. For this to succeed, the filtering key must be, not only different, but also enhanced compared to the blocking key. Instead of a 4-length prefix (which is the blocking key), for example, filtering could use a 5-length prefix or a complete word as a key.

Blocking and filtering do not exclude each other from the ER process, but they cooperate towards improving ER efficiency. In a sense, Blocking performs a grouping in data which incurs both false positives and false negatives, so effectiveness is slightly affected, while filtering eliminates true negatives, so efficiency is further improved (Papadakis, et al., 2019).

3. Selected Method

The basic approach is the so-called Traditional Blocking. According to this method, some carefully selected attributes are used separately or in combination to compose the blocking key. All records that share the exact same key are placed in the same block. While traditional blocking is remarkably accurate at locating matching records, it bears some serious disadvantages, especially when large datasets are involved:

- The exact matching condition used to allocate records in blocks, leads us to select a single attribute or a small subset of attributes as the blocking key components. Also, the selected attributes are usually the ones considered as somewhat generic (postal code, gender) and not the highly personalized ones (last name). Even though this practice aims at raising the matching accuracy rate, it also results in large sized blocks, therefore in a significantly high number of pairwise comparisons.
- The exact key matching condition is expected to miss record matches due to existing flaws of any kind within the data (misspellings, errors, missing information etc.).

In this work we study an improved version of Suffix Array Blocking (Aizawa and Oyama [2005]), a method which deals effectively with the issues of traditional blocking. Suffix Array Blocking stands out for its speed and performance, though it may raise some questions regarding accuracy, depending on the datasets and the fields composing the blocking key.

3.1 Suffix Array Blocking

Suffix Array Blocking (Aizawa & Oyama, 2005) follows the steps of traditional blocking in relation to attribute selection and the composition of the initial blocking key or the BKV. However, the BKV is not directly used for the distribution of records into blocks. Instead, a group of additional keys is derived from the BKV to apply partitioning. In fact, the BKV is analyzed to its suffixes, with character length greater than a predefined minimum length l_{min} and each one of these suffixes serves as the key to a separate block. For example, let there be a dataset containing personal information (last name, first name, home address, phone) and after thorough investigation we decide on a blocking key which consists of last name and first name values. For the initial blocking key value of "JohnThompson" and *minimum suffix length* l_{min} = 4 the suffix array generated contains the values showed in Table I:

Table I: Generated suffixes of minimum length l_{min} = 4, from "JohnThompson" BKV

JohnThompson
ohnThompson
hnThompson
nThompson
Thompson
Hompson
Ompson
mpson
pson

Now, all suffixes generated from each records initial blocking key are then inserted in a central indexing structure, sorted in alphabetical order. This indexing structure serves as an inverted index (inverted Indexes are used in cases where values are used to locate keys). Every suffix has a reference to a record in the dataset. This is the record which generated the corresponding BKV, the suffix is derived from. An example of such an index, containing suffixes from initial blocking keys "JohnThompson", "JohnThomson" and "BartSimpson" is shown in Table II:

Table II: Inverted Index structure, sorted alphabetically

Suffix	Record No
artSimpson	3
BartSimpson	3
hnThompson	1
hnThomson	2
hompson	1
homson	2
impson	3
JohnThompson	1
JohnThomson	2
mpson	1
mpson	3
mson	2
nThompson	1
nThomson	2
ohnThompson	1
ohnThomson	2
ompson	1
omson	2
pson	1
pson	3
rtSimpson	3
Simpson	3
Thompson	1
Thomson	2
tSimpson	3

Using the Inverted Index structure, we query each suffix (which also serves as the blocking key), retrieve all its referenced records and group them together within the same block. Now, each block contains references to the records related to the respective blocking key (suffix). According to Table II, suffixes "mpson" and "pson" include in the respective blocks both records that generated "JohnThompson" and "BartSimpson" BKVs. Another important feature is the fact that each distinct suffix displayed in Table II, corresponds to a block, therefore a record (e.g., Bart Simpson) referenced by more than one suffixes, is placed within multiple blocks. However, this redundancy is something we need.

Depending on the popularity of each suffix within the examined dataset, blocks might differ in size, more or less. A maximum block size parameter (l_{mbs}) is also defined to avoid highly sized blocks, a situation which may adversely affect efficiency. If the number of records referenced in a block exceeds the l_{mbs} value, then the block must be removed. This is a necessary precaution to protect the method's efficiency, since there are some quite popular suffixes (like *-ing*) that are expected to generate huge blocks, filled with not so relevant entities, leading to many unnecessary comparisons.

The l_{mbs} parameter along with a careful choice of the l_{min} value protect the technique's efficiency, without risking effectiveness. Besides, blocks defined by low length but rare suffixes remain valid. At the same time, the information in removed (large) blocks, can still be retrieved through other blocks, since the multiple suffixes of a blocking key allow a significant but useful number of redundant blocks.

For instance, let's examine what happens in a situation with the following BKVs:

BKV
biome**dical**
metho**dica**l
parame**dical**
paro**dical**
perio**dical**
ra**dical**
syn**dical**

Let $l_{min} = 4$ and $l_{mbs} = 6$

Each one of these seven BKVs generates the following identical suffixes: {**"ical", "dical"**}

This means that two blocks are created, with "**ical"** and **"dical"** as blocking keys respectively, and these blocks contain seven records, each. Since block size cannot exceed the $l_{mbs} = 6$ limit, these two blocks are eventually purged. However, all the above records continue to participate in the matching stage, since they are also referenced by other suffixes, that still remain active as blocking keys.

In general, Suffix Array blocking proves itself to be efficient due to several reasons:

- The potential matching records are grouped into rather small blocks. These blocks have also a high degree of relevance.
- The Suffix Array blocking algorithm is low in complexity, in contrast to more traditional methods
- Any value errors that may exist at the beginning of a BKV do not affect the efficiency of the technique. The corresponding records are placed into a proper block with a smaller suffix as a key. Thus, any redundancies produced with this method offers a powerful advantagewith respect to the effectiveness.
- The suffix logic gives us the opportunity to work with attributes combinations in order to define blocking keys. As a result, the blocks produced are quite smaller than the ones that would be created by single attribute keys. For example, in a very large dataset, a key value "John" (first name attribute), probably refers to a lot more records than a key value "JohnThompson" (first/last name attributes combination). More important, the records referenced by the latter key have a higher matching probability than those reference by the former one.

3.2 Improved Suffix Array Blocking

Nevetheless, Standard Suffix Array Blocking is not a perfect method, and it can reveal its vulnerabilities under specific conditions, such as the following:

- Since the suffixes generated are of length $\geq l_{min}$, all the suffixes of a BKV contain the last l_{min} characters of this key. This means that when an error occurs within the last l_{min} characters of the BKV (e.g., a spelling mistake), this error is inherited by all suffixes produced by this specific BKV. Therefore, matching records, whose BKVs differ due to such an error, do not have any matching suffix, and eventually are placed in different blocks.
- As is already mentioned, blocks containing more than l_{mbs} references to records are automatically removed from the process. A very important detail we need to keep in mind, is the fact that each suffix references at least the same number of records with longer suffixes, generated by the same BKV. Thus, when a suffix of length k is removed due to block size restrictions, all suffixes of the same BKV with length < k are removed as well. So far everything seems absolutely normal, however this is a case where the chances of achieving correct blocking via smaller suffixes may diminish significantly. More specifically, if a spelling mistake occurs at the beginning of a suffix of length k, which generates a large block, matching fails since all potential matching suffixes of length < k, (where the spelling mistake is present) are also excluded.

The improvement proposed by (Vries, et al., 2011) is based on the following:

1. A simple way of overcoming the problem of BKV value errors, which result in the placement of highly similar records in different blocks, is to group blocks with highly similar blocking keys (suffixes)
2. Suffix comparisons, regarding grouping, are performed using Jaro similarity function.
3. For this action to be efficient we must avoid, in general, an excessive number of comparisons between BKV suffixes:
    a. There is absolutely no point in comparing suffixes produced by the same BKV (as shown in Table 1, they are remarkably similar but differ only in the leading character).s
    b. The comparisons must be limited only between similar suffixes that are produced by different BKVs.
4. Since BKV suffixes are already sorted in the index list (Table 1), suffixes generated by the same BKV are scattered all through the list.
5. The applied sorting in the index list (Table 1), has brought most of the similar suffixes from different BKVs close or next to each other.

Having considered all the above, we can avoid unnecessary and ineffective suffix comparisons by grouping together blocks defined by suffixes that are neighboring in the index list. This can be applied with a sliding window technique, according to which suffix comparisons take place only within each window. The size of the sliding window may vary depending on the dataset's specifications. It is important, though, to remember that larger window sizes may offer higher accuracy, but they can also diminish efficiency at a notable level.

Therefore, Improved Suffix Array Blocking deals properly with the vulnerabilities of the standard method, especially when grouping is applied by the sliding window approach. In most of the cases where some spelling mistake causes slightly different BKVs, some of their suffixes end up close enough within the ordered inverted index structure, so there is a strong possibility of them being grouped together. Of course, there are cases where BKVs contain so many errors, that hardly any suffixes end up close enough to be grouped together. This probably means that the differences between these records' attributes are of such extend that they cannot be matched, anyway.

We should also mention the fact that this technique's complexity remains low (just like the Standard Suffix Array blocking technique). Even though the grouping stage is included, it seems that it does not have a significant effect on the total complexity of the procedure. Actually, in the worst-case scenario, where:

- The average BKV length is k
- The minimum suffix length $l_{ms} = 1$
- The number of records to match one another is n
- Every suffix is grouped together

the query time, for a single record, equals $O(kn \log kn)$. Of course, if normal datasets are involved, records are distributed into separate blocks, therefore the query time becomes $O(b \log kn)$ where b value is determined by the quality of the dataset. More potential matches cause higher b values. (Vries, et al., 2011)

A description of the studied algorithm, where the above procedure is defined, implemented by (Vries, et al., 2011), is shown in the following Table III:

Table III: Improved Suffix Array Algorithm

Improved Suffix Array Algorithm (Vries, et al., 2011)
Input:

1. $\boldsymbol{R_p}$ and $\boldsymbol{R_q}$: The sets of records to find matches between
2. $\boldsymbol{j_t}$: Similarity threshold for the suffix comparison function
3. $\boldsymbol{l_{ms}}$: The minimum suffix length
4. $\boldsymbol{l_{mbs}}$: The maximum block size
 	
Let $\boldsymbol{II}$ be the inverted index structure used
Let $\boldsymbol{C_i}$ be the resulting set of candidates to be used when matching with a record $\boldsymbol{r_{pi}}$
 	
//Index construction
For record $\boldsymbol{r_{qi}} \in \boldsymbol{R_q}$:
Construct BKV $\boldsymbol{b_{qi}}$ by concatenating key fields
Generate suffixes $\boldsymbol{a_{qi}}$ from $\boldsymbol{b_{qi}}$ where $\boldsymbol{a_{qi}} = \{\boldsymbol{s_{q1}}, \boldsymbol{s_{q2}}, .., \boldsymbol{s_{qy}}\}$,
$\lvert\boldsymbol{a_{qi}}\rvert = \boldsymbol{y} = \lvert\boldsymbol{b_{qi}}\rvert - \boldsymbol{l_{ms}} + \boldsymbol{1}$ and $\boldsymbol{s_{qj}} = \boldsymbol{b_{qi}}$.substring($\lvert\boldsymbol{b_{qi}}\rvert - \boldsymbol{l_{ms}} - \boldsymbol{j} + \boldsymbol{1}, \lvert\boldsymbol{b_{qi}}\rvert$)
For suffix $\boldsymbol{s_{qij}} \in \boldsymbol{a_{qi}}$:
Insert $\boldsymbol{s_{qij}}$ and a reference to $\boldsymbol{r_{qi}}$ into $\boldsymbol{II}$
//Large Block Removal
For every unique suffix $\boldsymbol{s_f}$ into $\boldsymbol{II}$:
If the number of record references paired with $\boldsymbol{s_f} > \boldsymbol{l_{mbs}}$:

Remove all suffix-reference pairs where the suffix is s_f

//Suffix grouping (Improved Suffix Array only)
For each unique suffix s_f into II (sorted alphabetically):
Compare s_f to the previous suffix s_g using the chosen comparison function (e.g., Jaro)
If $Jaro(s_f, s_g) > j_i$: (highly similar)
Group together the suffix-reference pairs corresponding to s_f and s_g using set join on the two sets of references

//Querying to gather candidate sets for matching
For record $r_{pi} \in R_p$:
Construct BKV b_{pi} by concatenating key fields
Generate suffixes a_{pi} from b_{pi} where $a_{pi} = \{s_{p1}, s_{p2}, .., s_{py}\}$,
$|a_{pi}| = y = |b_{pi}| - l_{ms} + 1$ and $s_{pj} = b_{pi}$.substring($|b_{pi}| - l_{ms} - j + 1, |b_{pi}|$)
For suffix $s_{pj} \in a_{pi}$:
Query II for a list of record references that match s_{pj}
Add these references to the set C_i (no duplicates)

3.3 Bloom Filters

So, with Improved Suffix Array Blocking we have made some progress regarding the reduction in comparisons between entities which probably do not match. However, a new issue has come up. Now, in contrast to the traditional blocking, each record's BKV generates several suffixes. This means that the inverted index is queried several times for each record that searches for matching candidates. This can cause some adverse effects on the process, since the index structure is stored on disk, which makes queries expensive, and especially when no results are returned, all this disk reading work is done in vain.

Generally, it is a fact that querying a dataset stored on disk, even if it is done only to check the existence of a record, comes to a cost. Then, when we just want to test a record's existence within a dataset, maybe we should look for a faster and a less expensive technique, one that does not have to query the dataset itself. Like having a checklist with guests, so when a guest arrives his name is checked before entering the building. This way we always know who are inside the building, just by checking the list. We do not have to shout names and expect someone inside to respond. This is where Bloom filters come of use.

A Bloom filter (Bloom Burton, 1970) is basically a data structure, organized in such a way that it can test whether an element exists in a collection of data or not. Although the conclusion is based on probability theories, the process proves to be remarkably reliable. A Bloom filter requires an array of bits, which serves as index for the elements stored in a collection. At first, while the collection is still empty, all bits within the array are initialized to zero (0) value. Every time an element is added to the collection, *k* hash functions with range equal to the array size are applied to its value (usually its string representation or in our case its BKV). All these functions return a value which points to a position inside the array of bits and, so the respective bit's value is turned to one (1) (Figure 4). This is the indexing phase of the process.

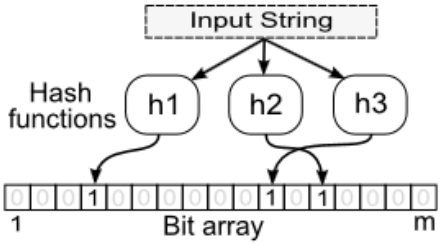


Figure 4: Bloom Filter implementation using k hash functions (Vries, et al., 2011)

There is, however, an alternative implementation which makes use of a single hash function with the addition of a random number generator. Instead of k hash functions to an elements BKV, only one hash function is applied, and the result is used as the seed for the random generator in order to generate k random numbers, which point to k positions inside the array of bits (Figure 5).

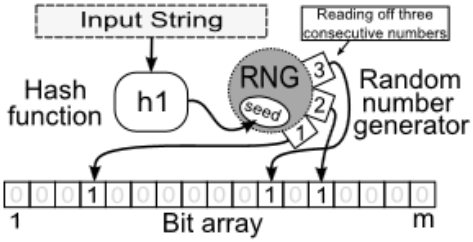


Figure 5: Bloom Filter implementation using one hash function and a random number generator (Vries, et al., 2011)

When we want to retrieve an element's information, before querying the collection we check if this element is inside it. The Bloom filter applies the proper hash functions on the examined element's value and compares the results with the bit values in the respective positions inside the array. If, and only if, all these bits have a value of one, then there is a high probability that this element is within the collection, therefore we can proceed by performing the query. Instead, when at least one bit has a value of zero, we are a hundred percent certain that the element is not present inside the collection, hence any query performed would be a lost cause. From the above, we can conclude that the Bloom filter process eliminates false negatives, while it is prone to false positives. However, even if a falsely positive outcome results in a useless query on the dataset, the total process' cost is raised to a reasonable extent and that is outweighed by the true negative results, which saves us the cost of not proceeding queries (Vries, et al., 2011).

Let n be the number of bits the Bloom filter array consists of and k the number of hash functions used. If n is the number of distinct elements inserted into the dataset, and therefore have invoked relative updates in the Bloom filter array, the probability r of a specific bit in this array to be set to 1 is given as follows:

$$r = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} \text{ (Bloom Burton, 1970)}$$

and the false positive rate p of the Bloom filter is:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k} = (1 - r)^{k} \text{ (Bloom Burton, 1970)}$$

4. Implementation Issues

In this work we aim at applying some modern techniques in order to make the algorithm in Table III more efficient. The anticipated performance boost can be achieved with parallelization, as when data are processed in parallel, they are distributed across different processing units (e.g., nodes of a cluster), reducing the time needed for the operations on the data to complete. So Improved Suffix Array Blocking with Bloom Filters algorithm is implemented both serially and in parallel, in order to conduct some comparative tests and demonstrate how different the two implementations are, in terms of performance. Regarding our parallel implementation of the Improved Suffix Array Blocking process, we create a Spark application written in Scala, while for the serial implementation, the application is written in Java.

4.1 Apache Spark

4.1.1 What is Apache Spark

One of the most popular frameworks for processing Big Data and the one we are using to perform our tests on Blocking and Filtering, in parallel, is Apache Spark. To understand what it is and how it came to life we need to make a historical review. When we refer to the efficient managing and processing of high volumes of data, a typical example that comes to our mind is search engines. Search engines, today, attain such speed regarding indexing and searching within data across the web, that once seemed inconceivable.

Specifically, one of the most powerful and efficient search engines is the one used by Google. It searches and indexes data in such a scale, that traditional tools, like relational database management systems or conventional storage systems are unable to handle. To accomplish this, Google developed its own arsenal of Big Data tools:

1. Google File System (GFS)
   This is an enhanced distributed file system, also fault-tolerant, able to handle large amounts of data

2. Bigtable
   This is a distributed system, built on GFS, to handle the storage of semi-structured data, and it can scale to a very large extent

3. MapReduce (MR)
   This is a framework, that offers programmers the opportunity to write relatively simple programs that handle reliably and in-parallel huge volume of distributed data across GFS or Bigtable. An MR application, essentially, exploits the MR system, so that the part of the program responsible for complex computations on the data is executed directly on the location of the data. This way, data distributed over the network is minimal, hence MR benefits significantly in terms of performance improvement. A simple description of an MR process is the following. Since large datasets are not

stored in a single location but are spread across clusters, MR uses cluster workers which take over the process of computations in an aggregate way. At the same time, the results of these computations are written to a file, also somewhere within the distributed file system, and is directly accessible to the MR application.

Google, by publishing three whitepapers, regarding the above technologies ("Google File System" (Ghemawat, et al.,), "MapReduce: Simplified data processing on large clusters" (Dean & Ghemawat, 2004) and "Bigtable: a distributed storage system for structured data" (Chang, et al., 2006)), inspired the development of Hadoop file system (HDFS), which became after some time part of a complete framework, known as Apache Hadoop. Apache Hadoop's purpose (like Google's tools) is to make available several tools that allow simple programs to handle and process, in a distributed manner, exceptionally large sets of data, across clusters of computers.

Despite its initial impact, Hadoop gradually begun to lose ground for a number of reasons. It seemed a bit difficult to handle, its MR API felt annoyingly verbose, it demanded a lot of setup code, but mostly, consecutive MR jobs required the intermediate computation results to be written to local disks, causing serious performance issues, due to heavy disk I/O. At the same time, new trends in Big Data world, like Machine Learning or Streaming, lead to the development of a new set of tools for Hadoop (Apache Hive, Apache Storm, Giraph and others), so that these new workloads could be handled. Nevertheless, this was not enough and new ways of improving Hadoop and MR were examined (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020).

This new challenge was undertaken by researchers who had already worked on Hadoop MapReduce and after a while project Spark came to life. Simply put, Spark had to be simpler, faster and easier than Hadoop MR, hence its development focused on system enhancement regarding the following:

- High fault tolerance
- Parallel in an exceptionally large extent
- In-Memory storage for intermediate results between consecutive computations
- APIs in multiple languages
- Support for other workloads

According to Apache, Spark is an open-source unified analytics engine, designed for large scale data processing (https://spark.apache.org/, n.d.). It includes a number of libraries for Machine Learning (MLLib), SQL for interactive queries (SparkSQL), stream processing for interacting with real-time data (Spark Srtream) and graph processing. Some of its distinctive features are the following:

- Spark accomplishes high speed by fully exploiting the latest years' hardware development. On the one hand multicore CPUs facilitate multithreading and parallel processing, while on the other hand the availability of hundreds of gigabytes of memory, helps in preserving intermediate computation results in memory, instead of writing them on disks.
- Spark offers simplicity in programming applications, mainly by providing developers with its fundamental data structure, the so called Resilient Distributed Document (RDD), a fault-tolerant collection of records managed in-parallel. Even more, several types of data structures may derive from an RDD, serving any kind of requirements.

Also, there is a quite sufficient number of operations that can be applied on these data structures, known as transformations or actions and they are available in many programming languages.

- Spark is able to read data from any external source available, so its mechanism is mainly focused on computations, which are performed in-parallel and in-memory, making the process particularly fast.

4.1.2 How it works

Apache Spark is fast and resilient. It can perform computations in a large amount of data, accomplishing both high speed and data consistency. But, beyond contemporary hardware improvement (CPUs, memory, multithreading, …) it is all about how to benefit from this by dividing the whole process in tasks and coordinating them optimally. Spark engine consists of some key components which work in a synergistic manner on a cluster of computers, to achieve distributed processing.

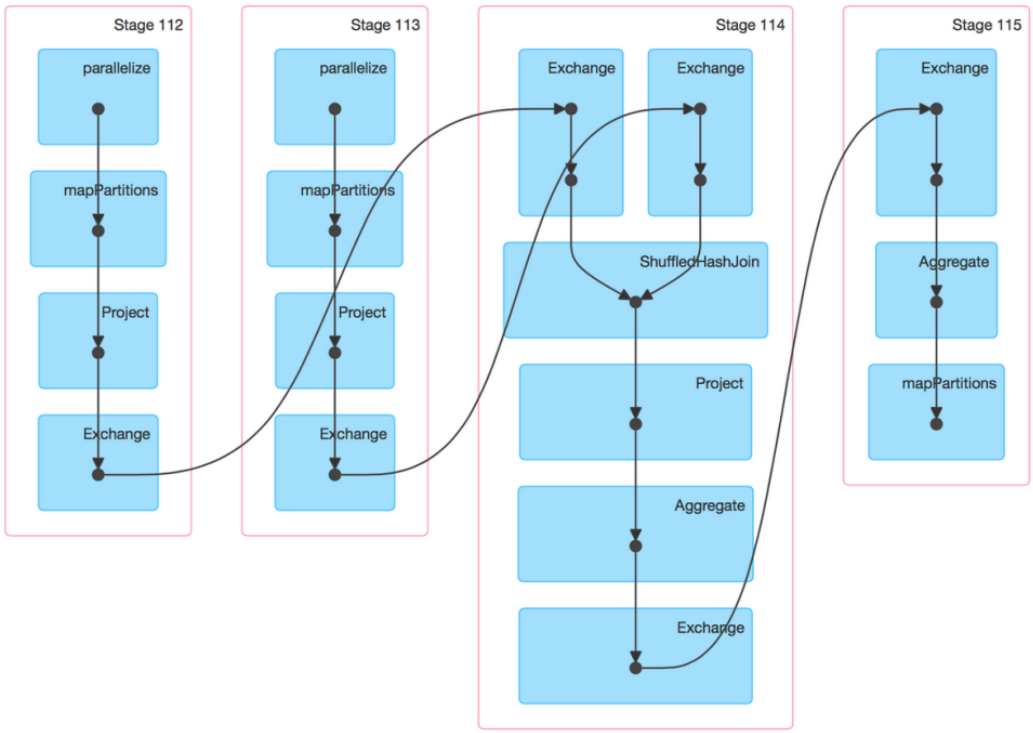


Figure 6: Spark's example of a Directed Acyclic Graph (DAG)

Spark makes use of the so called Directed Acyclic Graph (DAG). DAG contains many vertices which represent a stage or a task and edges that connect each vertex with the next in line, depending on the execution schedule (Figure 6). Since every Spark program includes a number of query computations, DAG scheduler and query optimizer generate a proper computational graph that can be broken into tasks in such a way, that they can be executed in parallel. So, if this program is executed on a cluster of computers, these tasks can be distributed across computers that play the worker role. However, for this to work, data must also be distributed

across cluster nodes and Spark has an exceptionally efficient and resilient way of partitioning data. In addition, Spark's execution engine which improves memory and CPU usage for Spark applications, as well as the fact that all intermediate computation results are kept in memory, enhance its performance, remarkably.

In order to take advantage of Spark, an appropriate application must be created. A Spark Application is based on Spark APIs and is distinguished in two main parts, the driver program, and the executors. To be able to access Spark functionality, an application needs some sort of intermediary object, which in our case is a Spark Session object, created inside the driver program. Then Spark reorganizes the application and generates a number of Jobs, which basically are computations that can be analyzed further into Tasks and therefore are organized into a DAG. Some of the job tasks can be executed in parallel, while others must follow a serial execution plan, thus Spark organizes job operations in Stages. So, in a nutshell, a Spark Application, through its Driver Program, gains access into Spark APIs, obtaining the functionality of analyzing the program, acknowledging the necessary transformations, creating the proper Jobs, which, in turn, are organized into Stages comprising of Tasks (Figure 7). (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020)

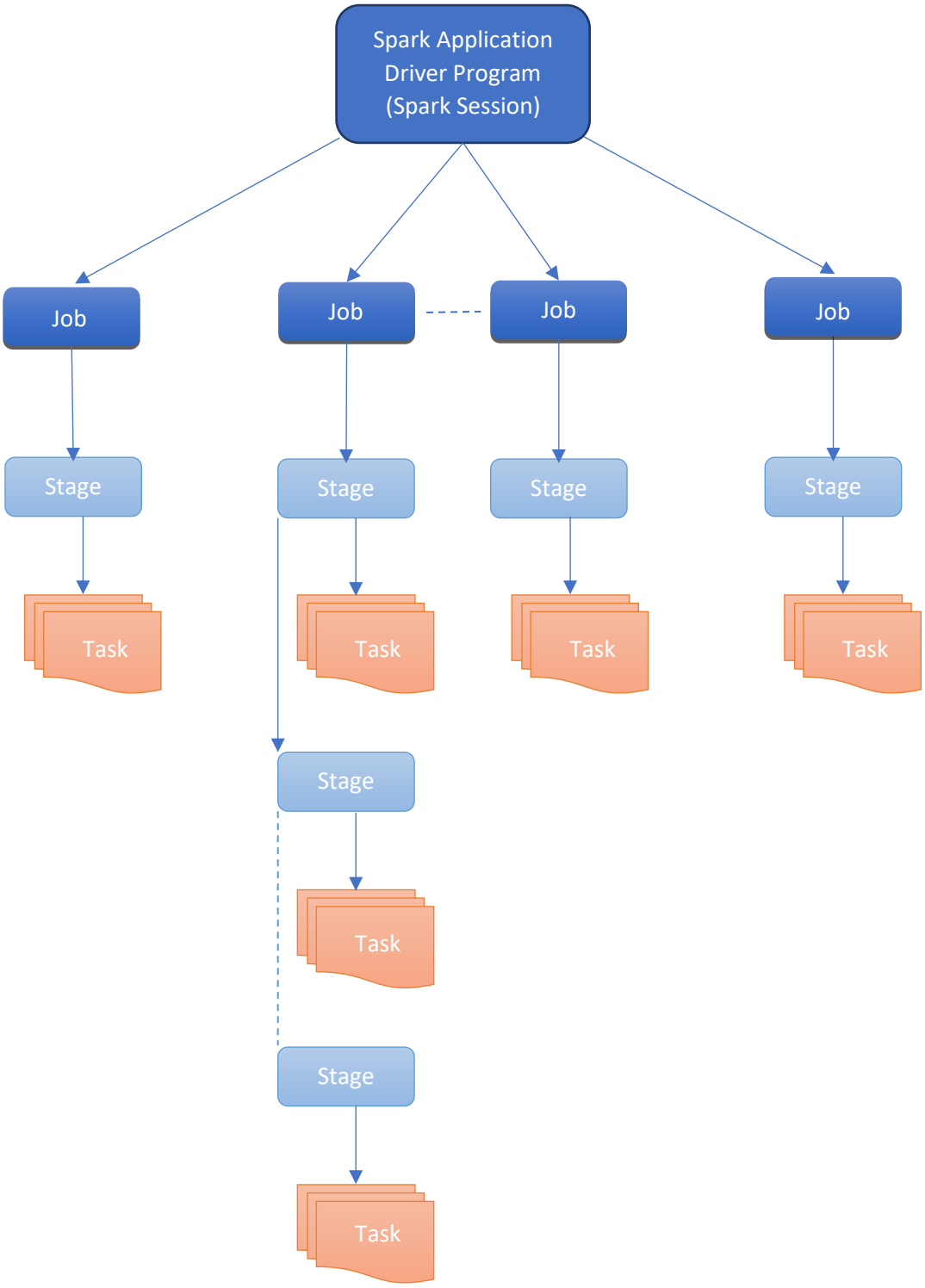


Figure 7: Spark Driver creating Jobs, Jobs creating Stages and Stages creating Tasks

4.1.3 Spark's Distributed Architecture

So, to put it simply, if we want to process a large amount of data, all we need to do is create an uncomplicated program that performs some tasks upon the data, and then we let Spark decide how to distribute these tasks among a cluster of computers, effectively and quickly (Kane, Frank, 2017). To understand how this works we must take a look at the individual components which constitute Spark's distributed architecture.

The simple script we write (in Scala, Python or Java) is, actually, the Driver Program, and inside the driver program reside the Spark Driver and the Spark Session. Another important component is the Cluster Manager (Hadoop, Yarn, Kubernetes, Standalone Spark Cluster), which serves as an orchestrator system. Cluster Manager is responsible for managing the hardware resources and distributing the work properly. Its purpose is to find out where all those different jobs must be assigned, in order to achieve the optimal result. And finally, there are the Spark executors, which run on the cluster worker nodes and ensure that tasks are executed, as scheduled.

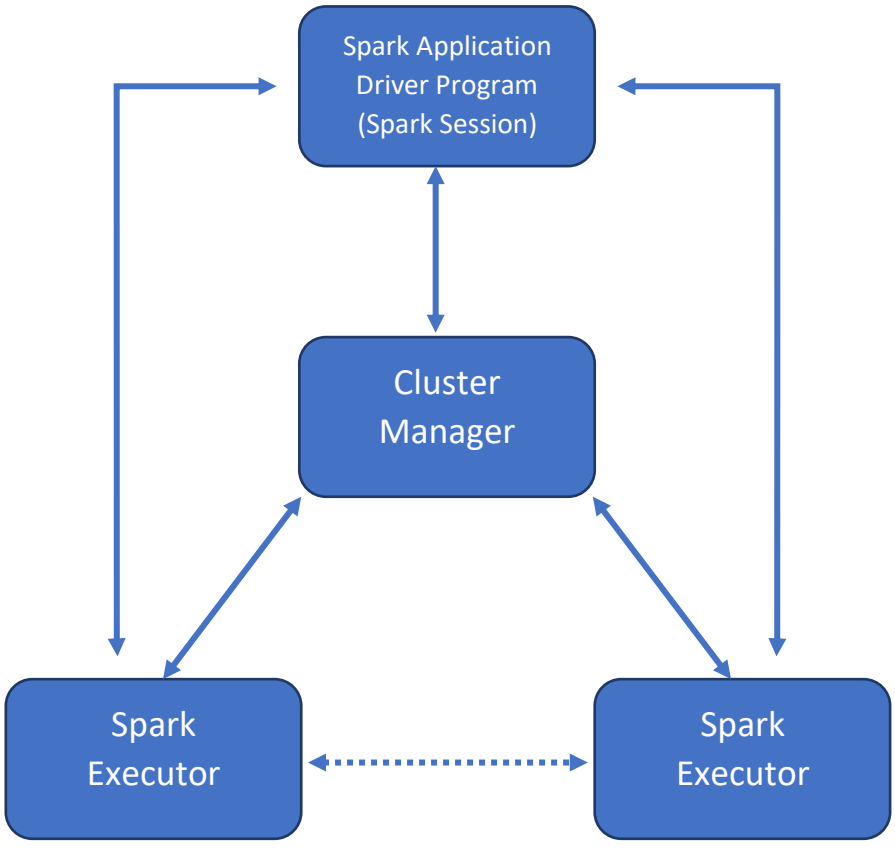


Figure 8: Spark's distributed architecture (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020)

All the above components (Driver Program, Cluster Manager, Executors) are in constant communication with each other. The Driver Program addresses the Cluster Manger but also directly the executors. The Cluster Manager communicates with everyone, decides what to run, where and when and gathers the partial results composing the final result. Finally,

Executors talk to each other, so they remain synchronized. (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020) (Kane, Frank, 2017).

4.1.4 Spark RDDs, DataFrames and Datasets

As mentioned previously, Spark breaks down an application in smaller units, that can be executed in parallel. A specific computation, for example, on a dataset is assigned to a Stage and the Stage, depending on the number of data partitions, assigns the computation on each partition, to a Task. Eventually, a Task is a computation applied on a single partition of data, and when all Tasks of a Stage are completed, then we can assume that the specific computation, the stage has undertaken, has completed. In order for this to proceed, the data must be in a distributable and resilient form.

RDD (Resilient Distributed Dataset) is the main abstraction (more particularly, a distributed memory abstraction), provided by Spark. It is a collection of elements of data, it is immutable and can be distributed across cluster nodes and be processed in parallel. In fact, distribution and parallelization is the reason of the immutability of Spark data components. Immutable data can be partitioned safely and easily since they exclude concurrent modifications via multiple threads.

There are two types of operations that can be applied on an RDD. *Transformations* and *Actions.* A *transformation* is the operation of generating a new dataset, by applying some modifications or computations to another one (e.g., applying a *map* function on an RDD, creates a new RDD). An *action* is the operation of returning the result of some computations on an RDD, to the driver program (e.g., *collect* the results of a transformation). We should point out the fact that an action call is what triggers the execution of transformations.

RDDs are recomputed every time they are called, so in a way, they are more of a set of directions on how to transform data, than transformed data themselves. However, in cases where multiple transformations may occur on the same dataset, Spark offers the option of caching and persisting an RDD in memory, in order to be used efficiently for parallel operations. (https://spark.apache.org/, n.d.)

DataFrames are also immutable and distributed data collections, but they are organized in schemas, have named columns, as well as a definite data type for each column. One could say that a DataFrame resembles a database table. This structural form of data offers many conveniences on managing and operating on Spark's DataFrames. Datasets, on the other hand, are strongly typed immutable and distributed data collections. A DataFrame is practically an untyped view of a Dataset (Dataset of Rows). Operations like transformations and actions are available on Datasets and DataFrames. As in RDDs, an action must be called to invoke the corresponding transformation (Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020).

4.2 Scala

Scala is an object oriented as well as a functional programming language. Its development started in 2001 in Lausanne, Switzerland by Martin Odersky and became publicly available in 2004. It has evolved significantly since then, and over the last years it grows in popularity. Its name stems from the phrase **Sca**lable **La**nguage, and this is due to the fact that it was designed, from the beginning in manner of scaling up according to the programmer's needs. As a functional programming language, it offers the advantage of creating powerful programs from simple components and it is suitable for writing small scripts to perform some individual tasks. On the other hand, as an object-oriented language, it proves convenient for larger systems construction. It interoperates seamlessly with Java, since it runs on the Java platform, and can exploit all Java libraries (Martin Odersky, Lex Spoon, Bill Venners, 2011).

Scala's syntax is rather compact and resembles that of Ruby or Python (No semicolons or type annotations). Many of Scala's components are library abstractions, providing the programmer the ability to extend and adapt them according to her demands. It may not offer all possible constructs one might need; however, it offers all the necessary tools to crate them. For example, the interoperability with Java, combined with Scala's adaptive philosophy, allow the construction of new Scala objects by wrapping Java classes (Martin Odersky, Lex Spoon, Bill Venners, 2011).

Scala is a pure object-oriented language. Every value that is created within a program is indeed an object and every operation performed is an object's method call. This is somehow different from other object-oriented languages, where primitive values, for instance, are not objects themselves or static methods that do not belong to any object are permitted.

Being, also, a fully functional language, Scala has the following properties:

- Functions are values of the same status
- Functions can be passed as arguments to other functions
- Functions can return other functions as a result
- Functions can be nested
- Functions can be anonymous
- Scala encourages immutability, which makes it ideal for programs executed in parallel

Since Apache Spark is itself written in Scala, this is something we have considered, in order to decide which programming language, we are using for the parallel implementation. Choosing Scala for Spark Application offers us a somewhat better performance, but not in a huge degree. There are indeed some key differences between Java and Scala, but since they both run on JVM, their performance cannot be that different. Besides, Scala is designed to be interoperable with Java and can use any Java library or framework, because of the JVM. Some of the differences between Java and Scala are the following:

- Java requires longer lines of code, as well as a lot of boilerplate code.
- Java on the other hand, because of the detailed, code is less complex than Scala.
- Scala is a functional programming language and in fact it was designed as one, while Java introduced functional programming in version 8.
- Scala enforces code in an immutable manner, which makes easier to apply parallelism.

Nevertheless, as a general assumption, Scala is faster than Java and this is probably due to the fact that Scala is compiled into bytecode faster than Java. Scala's significantly better performance though, comes when we are dealing with parallel execution, so regarding a Spark application, the Scala language obviously has the advantage, over Java, in terms of both performance and ease of syntax.

4.3 Implementations

In this work, we focus on Data Linkage, the situation where we bring together data from different sources creating a more complete dataset. However, for this dataset to be functional, we have to match records between the separate ones. So, we create matching blocks following the Improved Suffix Array Blocking procedure. We may consider that two separate datasets need to be consolidated, the reference dataset, and the linkage dataset.

The algorithm distinct stages are the following:

1. Index Construction
    - Read the reference dataset.
    - For every record, create a Blocking Key Value (BKV) by concatenating the predefined record fields.
    - Then, for every BKV generate all possible suffixes, longer than minimum suffix length value.
    - Create Inverted Index structure, by inserting every suffix and a reference to the record it derives from.
2. Large Block Removal
   For every unique suffix within the Inverted Index Structure, if the referenced records are more than the maximum block size value, remove the suffix and all its references.
3. Suffix Grouping.
    - Sort Inverted Index structure and check If adjacent suffixes are similar (according to Jaro Similarity method and a predefined similarity threshold).
    - If so, then merge records referenced by similar suffixes in the same block.

   However, there is some ambiguity on how to handle this stage. For example, let x, y, and z be three consecutive suffixes within a sorted Inverted Index. Similarity comparison is performed between x-y and y-z suffixes. Now, we can approach this situation in two ways:
    a. Comparisons (x, y) and (y, z) are handled completely independently, that is if x-y blocks are merged and y-z blocks are also merged, elements of block x and elements of block z do not exist in common blocks.
    b. If comparison (x, y) results in merging x-y blocks, then the new xy block is the only one referenced by both x and y suffixes, so the next comparison between y and z suffixes (if found similar), end up merging block xy with block z. In the end if x is similar to y and y is similar to z, all elements from blocks x, y, z end up in the same block.

   In this work we have followed the second approach (b).
4. Create and fill out the Bloom Filter with the Inverted Index keys (suffixes)
5. Query Inverted Index for candidate sets for matching.

- Read the linkage dataset, and for each record, create a Blocking Key Value (BKV) by concatenating the predefined record fields.
- For every BKV generate all possible suffixes, longer than minimum suffix length value.
- For every suffix, query the Bloom Filter, and if there is a positive response, query the Inverted Index, get the list of referenced records by this suffix and create a matching set for the currently examined linkage record.

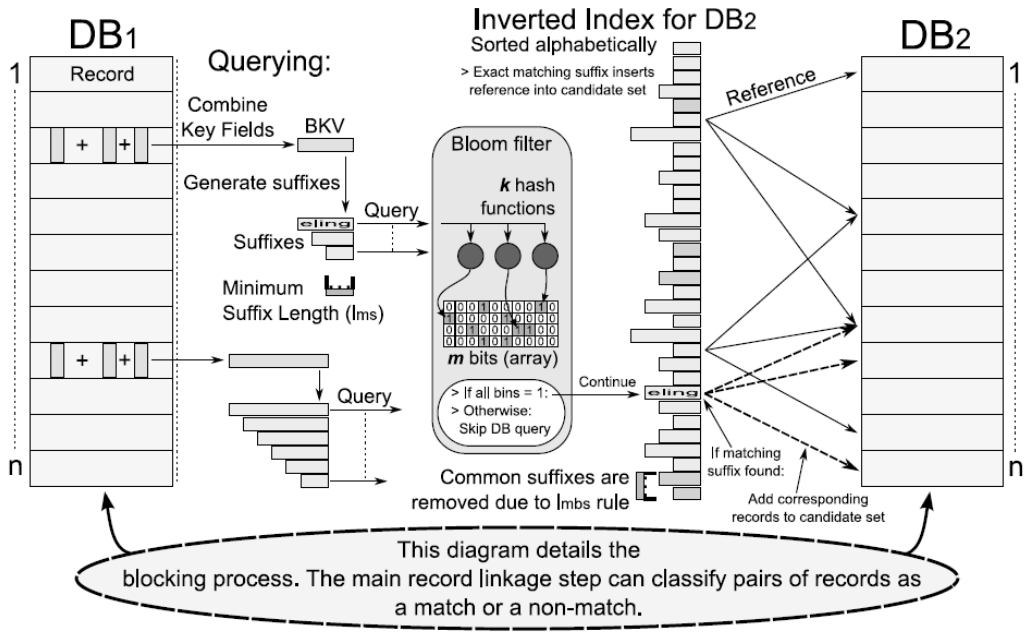


Figure 9: Overall Improved Suffix Array Blocking approach with the addition of a Bloom Filter (Vries, et al., 2011)

4.4 Serial Implementation

Our implementation for the serial execution of the Improved Suffix Array Blocking process is a program written in Java. Also, because in this work we study the possible advantages of parallelizing ER methods for Big Data, even though our experimental datasets do not have the size to be considered Big Data, our implementation must follow the plan of dealing with Big Data. Therefore, we cannot, for example, load the whole dataset in a Java ArrayList, because depending on the data volume and the available hardware resources, this approach may cause corresponding exceptions. Instead, we keep the data in a storage system.

In this case, we store our datasets in a PostgreSQL relational database and we use the following tables:

- A table for the reference dataset.
- A table for the linkage dataset.
- A table for the temporary version of the Inverted Index, before the grouping stage. In this table, each suffix is related to the BKV and the record, it is derived from.

- A table for the Inverted Index, after the grouping stage. In this table, each suffix is related to the record, it is derived from, but also to a block number.
- A table to store matching blocks. In this table, each record is related to a block, and records from the linkage dataset are also stored here. In the Matching phase (not implemented in this work) records associated with the same block are checked if they refer to the same real-world entity.

We enhance query performance by adding indexes where it seems necessary.

The algorithm stages, adjusted accordingly for the Java implementation, become as follows:

1. Connect to database and start reading data from the reference dataset table. For each record:
    a. Concatenate predefined fields to construct BKV.
    b. Generate all possible suffixes from BKV.
    c. Store each suffix with a reference to the BKV and the record id it is derived from in the temporary inverted index table.
2. Count references for each suffix within the temporary inverted index table and if more than maximum block size value is found, delete all corresponding suffix – records references.
3. Read from (sorted by suffix) the temporary inverted index table and compare each record (current) to the previous one, performing a similarity check on suffix field. The comparison takes place only if suffixes differ and were generated from different BKVs. If suffixes are found similar, then all current records are updated to refer to the same block as the previous suffix.
4. Fill out the Bloom Filter with suffixes from the Inverted Index table.
5. Read data from the linkage dataset table. For each record:
    a. Concatenate predefined fields to construct BKV
    b. Generate all possible suffixes from the BKV
    c. For each suffix, query the Bloom Filter. If a positive response is given, then query the Inverted Index table and collect the referenced records, creating a group of possible matches for this linkage record.

4.5 Parallel Implementation

According to the description of the implemented algorithm (Table III), this blocking method is configured to be applied to a record linkage case, such as when we want to consolidate two different datasets. The first part of the algorithm, essentially, creates an index after processing all the data inside a specific and finalized dataset, the so-called reference dataset. Consequently, it is safe to assume that we are dealing with an immutable dataset and an immutable index. Likewise, the linkage dataset (the one that performs the queries on the index) is also finalized and apparently also immutable. Since immutable data simplify parallel processing, this algorithm is quite suitable for parallel implementation.

Regarding the potentially very large datasets, in the parallel approach, we manage things quite differently than in the serial implementation. Spark handles hardware inadequacies in a very efficient manner. In fact, there are no memory limits. According to Apache Spark, processed data that do not fit in memory, are spilled to disk. Even cached datasets, when they exceed

memory limits either they can be saved to disk or Spark may recompute them on the fly (https://spark.apache.org/, n.d.). So, for the parallel implementation, we use the database, only to read the input and store the output data. In the meantime, we are using Spark Datasets.

Accordingly, the algorithm stages for the Spark Scala implementation become as follows:

1. Connect to database and load input data in a Dataset.
2. Transformation(s): Concatenate predefined fields to construct BKVs.
3. Transformation(s): Generate all possible suffixes from each BKV.
4. Transformation(s): Create the temporary indexing structure.
5. Transformation(s): Create the Inverted Index after removing large blocks.
6. Transformation(s): Create the final Inverted Index structure after merging blocks, referenced by similar suffixes.
7. Create and fill out the Bloom Filter, from the final Inverted Index
8. Connect to database and load linkage data in a Dataset.
9. Transformation(s): Concatenate predefined fields to construct linkage BKVs.
10. Transformation(s): Generate all possible suffixes from each linkage BKV.
11. Transformation(s): Select only linkage suffixes that seem to exist in the Bloom Filter
12. Transformation(s): Create matching blocks by joining linkage suffixes to the Inverted Index
13. Store matching blocks in the database

Some helpful details:

- Every step of the process may include more than one transformation.
- Every step of the process, generates one or more Datasets, structured properly, so that next step' s transformations are as simple as possible.

5. Experiments

In our experiments, we aim at studying the differences between execution in parallel of Improved Suffix Array Blocking, enhanced with Bloom Filters (Figure 9), and serial execution. Measurements are taken upon the time needed for the process to be completed, as a whole, as well as individually for every step of the process. We should point out that our measurements are used to demonstrate any improvement concerning only the performance of the process. Besides, the algorithm is implemented as is, so its effectiveness is not affected. If we study the Improved Suffix Array Blocking (with Bloom Filter) algorithm, as described in the previous sections, we can see that it can be split in separate stages, which are considered as distinct sub processes. These sub processes are used to carry out measurements for the partial comparisons.

As input data, synthetic datasets are used in the experiments, which have been generated by the Febrl tool, with its default settings (Christen, 2008). To compare parallel and serial performance of the selected method and the scalability of each, we need to compare the results after experimenting with a variety of dataset sizes. More particularly, we have generated 6 datasets of different size:

1. 5k dataset: 2.000 original records and 3.000 duplicates, with a maximum of 5 duplicates for each original record
2. 25k dataset: 10.000 original records and 15.000 duplicates, with a maximum of 5 duplicates for each original record
3. 50k dataset: 20.000 original records and 30.000 duplicates, with a maximum of 5 duplicates for each original record
4. 125k dataset: 50.000 original records and 75.000 duplicates, with a maximum of 5 duplicates for each original record
5. 250k dataset: 100.000 original records and 150.000 duplicates, with a maximum of 5 duplicates for each original record

6. 500k dataset: 200.000 original records and 300.000 duplicates, with a maximum of 5 duplicates for each original record

These datasets consist of data representing people and the data is structured in the specified columns:

- rec_id
- culture
- sex
- age
- date_of_birth
- title
- given_name
- surname
- state
- suburb
- postcode

- street_number
- address_1
- address_2
- phone_number
- soc_sec_id
- blocking_number
- family_role

Our Improved Suffix Array setup is as follows:

1. Minimum suffix length, $l_{min} = 6$
2. Maximum block size, l_{mbs} = 12
3. BKV is composed by fields {surname, given_name}, in this specific order

Hardware specifications:

- CPU: Ryzen 5 3600XT (12 Logical Processors)
- RAM: 16GB, 3200MHz
- Disk: M.2 NVMe, PCI Express 3.0

Our database server is a local PostgreSQL Server (v.14)

Improved Suffix Array with Bloom Filters process can be represented in a form of stages:

1. Stage 1: Read data, construct BKVS and generate suffixes, forming the first version of the Inverted Index, then remove large blocks (removing blocks turned out to be a rather fast task and should not be evaluated separately)
2. Stage 2: Merge blocks
3. Stage 3: Read duplicate dataset and create matching blocks for each duplicate record

In order to be able to draw more accurate conclusions, apart from the overall process, we also measure the former stages separately.

5.1 Serial Execution vs Parallel Execution

In this section we present the results of serial and parallel execution of the Improved Suffix Array Blocking (with a Bloom Filter) process. Results for both implementations per stage are compared separately in addition to the overall process for all six generated datasets.

5.1.1 Stage 1

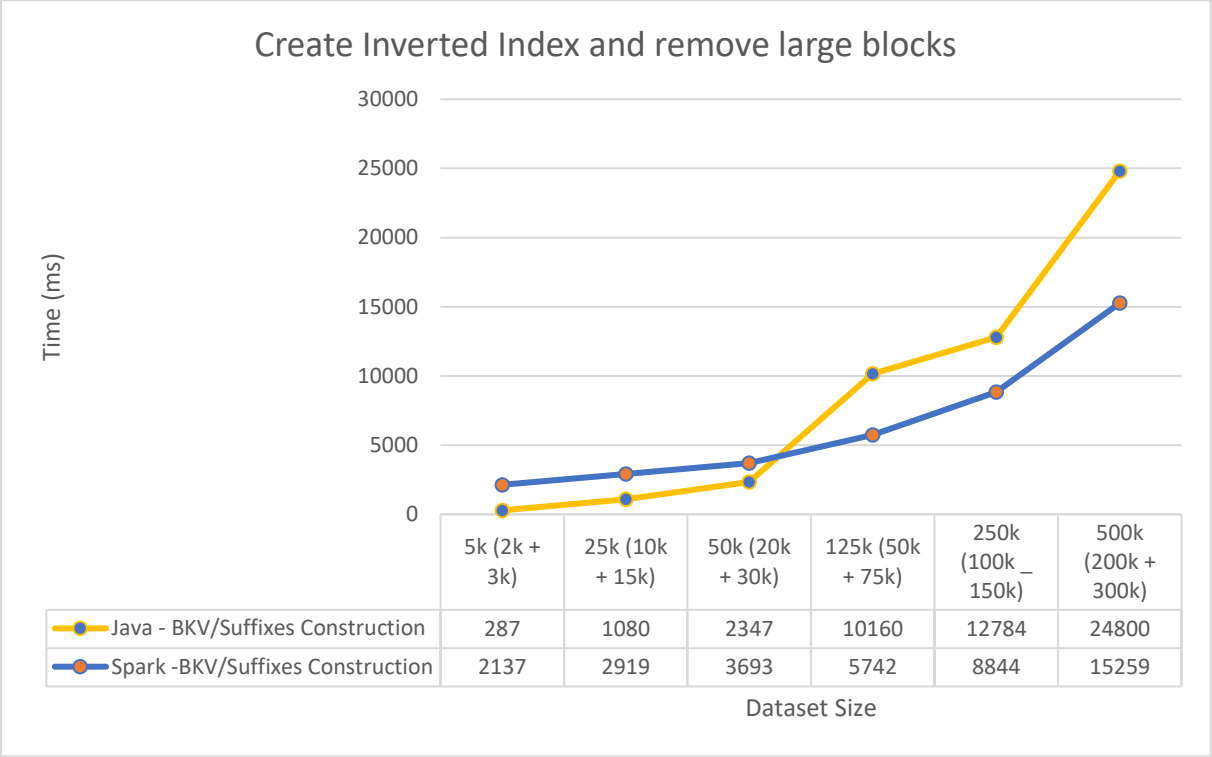


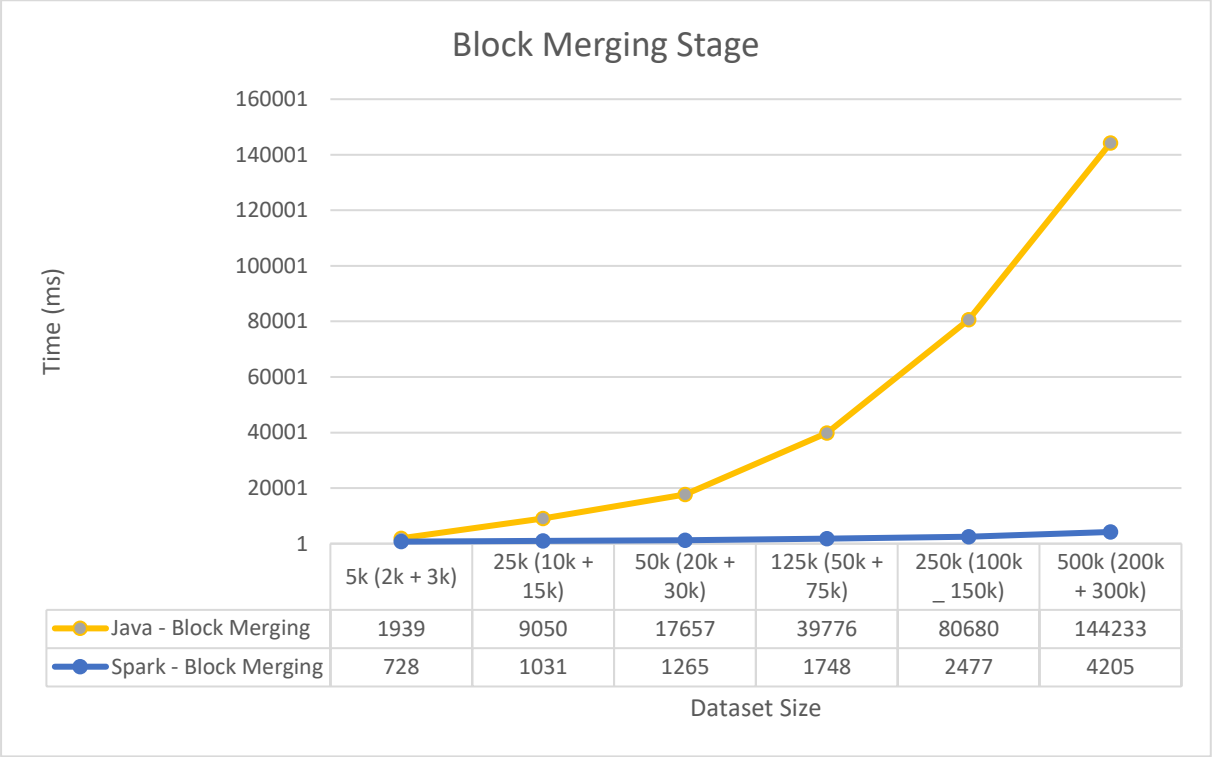
Figure 10: Stage 1 - Comparative performance graph

We first present results from Stage 1 of the process, that is: Read data – Construct BKVs – Generate suffixes – Build Inverted Index – Remove large blocks – Merge Blocks by similar suffixes.

We can see in Figure 10 that both serial and parallel implementations, show an increasing trend, regarding the execution time, according to the size of the dataset. However, we can focus on some interesting observations:

- For smaller datasets (5k, 25k, 50k), serial execution is faster (this is confirmed by carrying out the experiment, repeatedly). This could be an indication that distributed processing techniques may not be suitable for a limited size of data. This is not very strange though. Spark needs to do some work, under the hood, so the program can run in parallel (data division, store information about the divided data, decide where each data will be routed, collect results, recombine data, etc.) and this work takes some time. So, this amount of time can be quite noticeable when data processing completes very fast (in case of small datasets), but can be nullified, when data needs more time to be processed (very large datasets).
- Parallel implementation has a smoother increasing trend than the serial.
- For larger datasets, serial implementation shows a quite sharp upward trend

5.1.2 Stage 2



	5k (2k + 3k)	25k (10k + 15k)	50k (20k + 30k)	125k (50k + 75k)	250k (100k _ 150k)	500k (200k + 300k)
Java - Block Merging	1939	9050	17657	39776	80680	144233
Spark - Block Merging	728	1031	1265	1748	2477	4205

Figure 11: Stage 2 - Comparative performance graph

For Stage 2 that refers to merging blocks by similar suffixes, as we can see in Figure 11, the performed tasks are incredibly faster when executed in parallel and the difference in performance increases remarkably as the dataset size grows larger (even up to 30 times faster). This is a case where Apache Spark shows its potential very clearly. In fact, there is a quite logical explanation of what happens here:

- In the serial implementation, all data produced during intermediate calculations are stored (and therefore queried) in the database.
- Spark, on the other hand, stores all data coming from intermediate calculations in memory (and if memory is insufficient, it stores them locally on disk) and performs no query or transaction to the database. Simultaneously, Spark distributes tasks across cluster nodes, gaining even more in speed.
- During this stage, Spark remains faster, even with smaller datasets.

It seems that the traditional two-tier approach, where a client application performs a large number of transactions and queries in a database, while executing a job, no matter how satisfactory the execution time is, based on conventional criteria, has a hard time keeping up with a modern distributed approach.

5.1.3 Stage 3

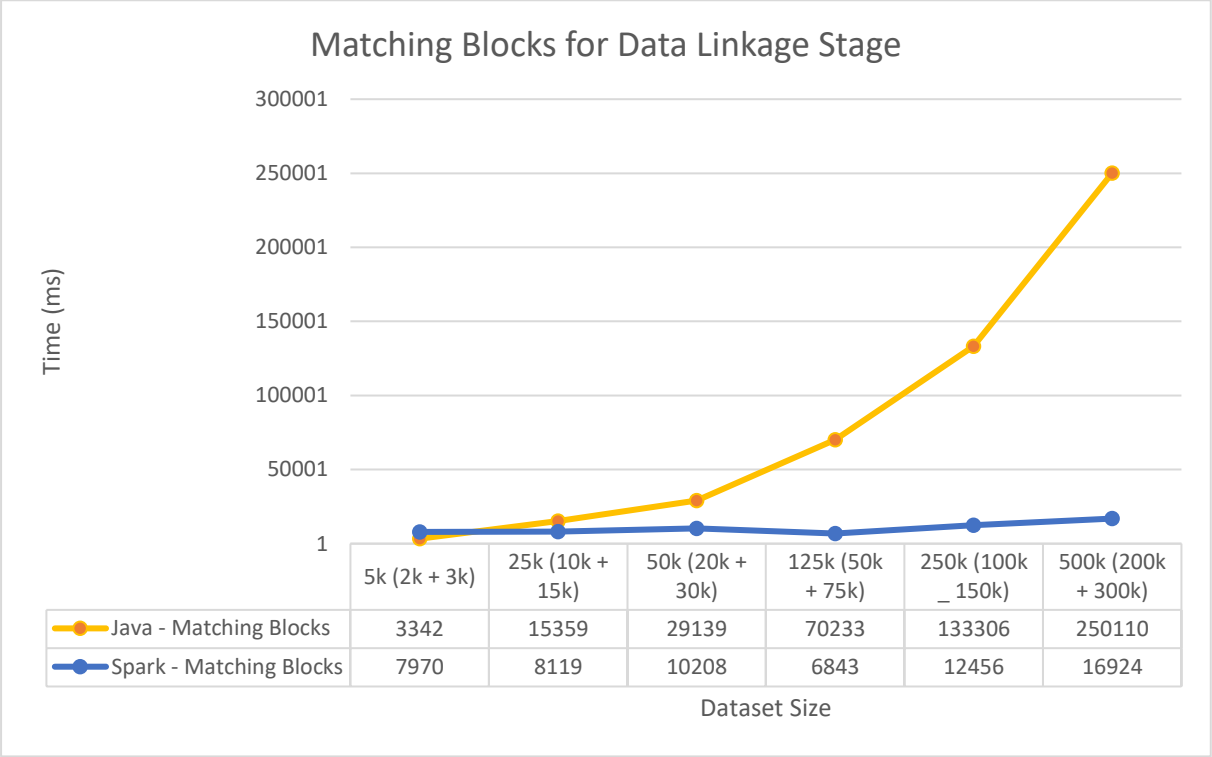


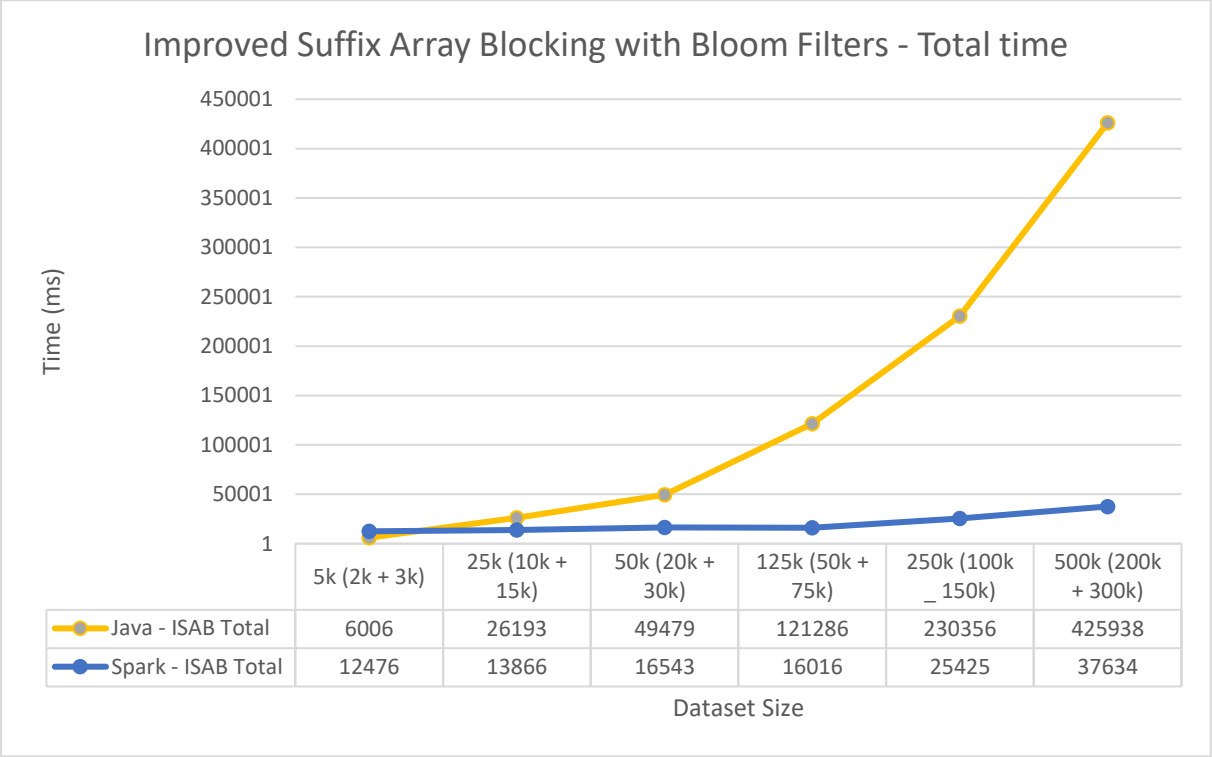
Figure 12: Stage 3 - Comparative performance graph

For Stage 3, Read duplicates dataset – Query Bloom Filter and Inverted Index – Create matching blocks, even though, at first glance, it seems that difference in speed between the two methods is similar to stage 2, a closer look in Figure 12 provides us with some interesting details:

- First, the difference in performance increases at a very large extend as the dataset size grows larger, like in stage 2
- As in Stage 1, Spark shows similar behavior when processing small datasets. Except from the fact that in the 5k dataset, serial application performs better, we notice that for datasets 5k, 15k and 50k execution time increases, until the 125k dataset, when execution is reduced remarkably, even lower than execution time the smallest dataset.

It is obvious that the larger the dataset the better the performance gains, when parallel processing techniques like Spark applications, are recruited.

5.1.4 Overall Process



	5k (2k + 3k)	25k (10k + 15k)	50k (20k + 30k)	125k (50k + 75k)	250k (100k _ 150k)	500k (200k + 300k)
Java - ISAB Total	6006	26193	49479	121286	230356	425938
Spark - ISAB Total	12476	13866	16543	16016	25425	37634

Figure 13: Overall Process comparative performance graph

By studying Figure 1, where the overall performance of the two methods is displayed, we can see everything we noticed by analyzing the results of the individual stages:

- A small advantage of Java application over Spark in quite small datasets.
- Spark's different behavior with smaller datasets and a boost in performance as datasets grow larger
- The enormous difference in performance, between the two methods, regarding very large datasets

It would be helpful to do some calculations and view the difference in performance in a more quantitative approach. The Spark application compared to the Java application is:

- ~ 2 times slower, regarding the 5k dataset (the smallest)
- ~ 2 times faster, regarding the 25k dataset
- ~ 3 times faster, regarding the 50k dataset
- ~ 7 times faster, regarding the 125k dataset
- ~ 9 times faster, regarding the 250k dataset
- ~ 11 times faster, regarding the 125k dataset

As far as Spark's performance with small datasets is concerned, the problem occurs in Stages 1 and 3. Considering the fact that at the beginning of the process (which is included in stage 1) Spark does all the necessary work to organize data distribution across nodes and parallel execution, while at the end of the process (included in stage 3) Spark collects results from all

the nodes and recombines datasets, our assumption of the reasons for this reduced performance, mentioned at Stage 1 analysis, seems logical.

5.2 Resources Usage

Another interesting view of the execution process is how each method exploits the available resources. How is the load shared across logical processors and how memory is utilized.

By observing Figure 14 and Figure 15 we are able to notice that while CPU has a lot of resources available, the Java application cannot exploit them because it is designed to run serially. Therefore, it utilizes only of a small percentage of the available CPU (< 18%).

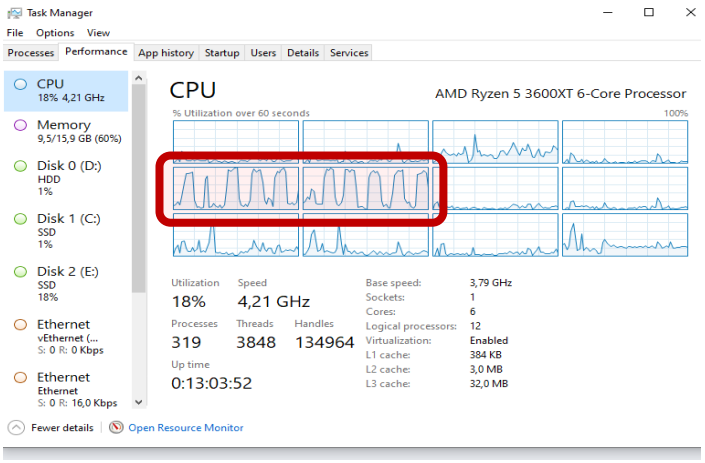


Figure 14: Serial Execution - CPU Utilization

Have in mind that this snapshot is taken at the most demanding part of the execution process, Stage 3 (linking data and creating matching blocks), where the application is performing queries and transactions on the database, so most of the CPU is used by the database server (Figure 15).

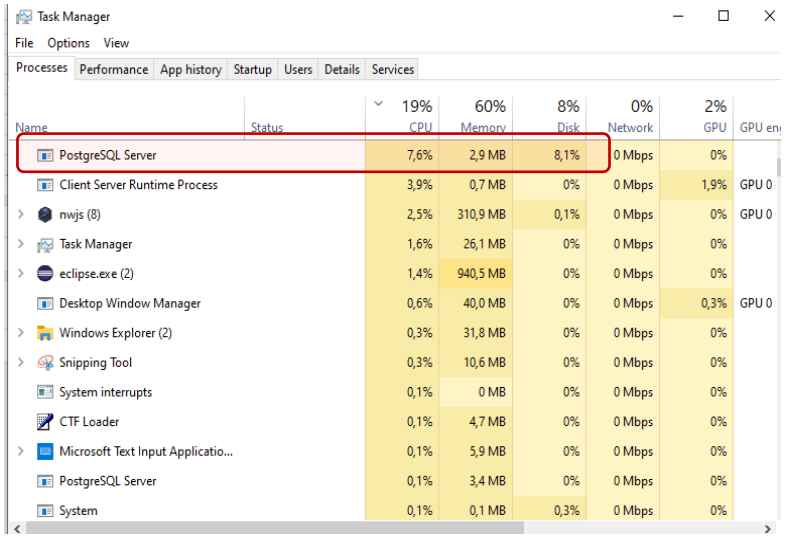


Figure 15: Serial Execution – PostgreSQL CPU and Memory Usage

Let us focus on Figure 16, to understand how Spark manages to perform so efficiently. This is a completely different kind of situation. Our program uses as many logical processors as possible and distributes the tasks accordingly. Now, CPU Utilization has reached over 40%, and as shown in Figure 17 (this snapshot was taken some seconds later), Spark application utilizes 90% of the total CPU usage.

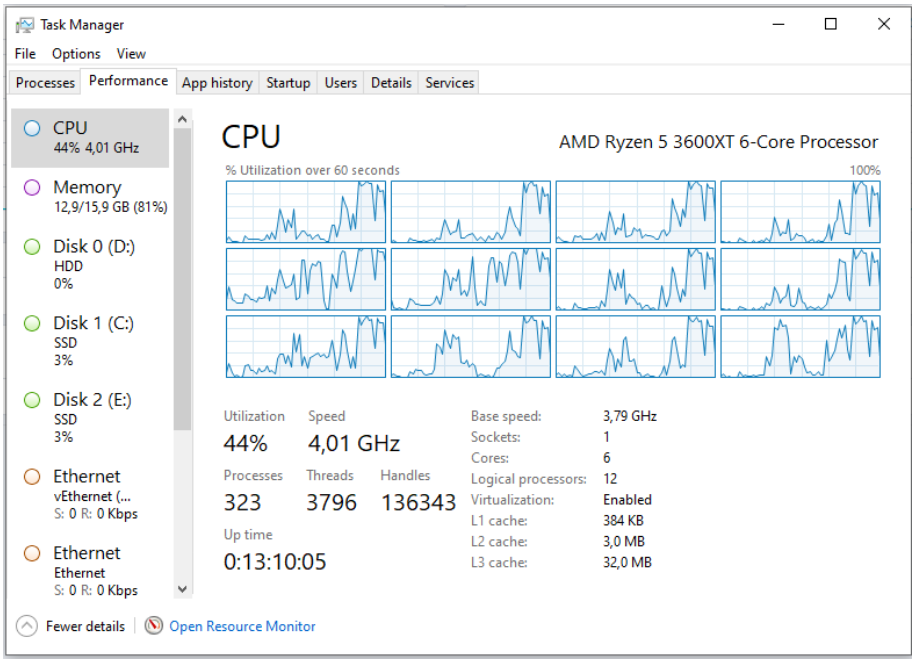


Figure 16: Parallel Execution – CPU Utilization

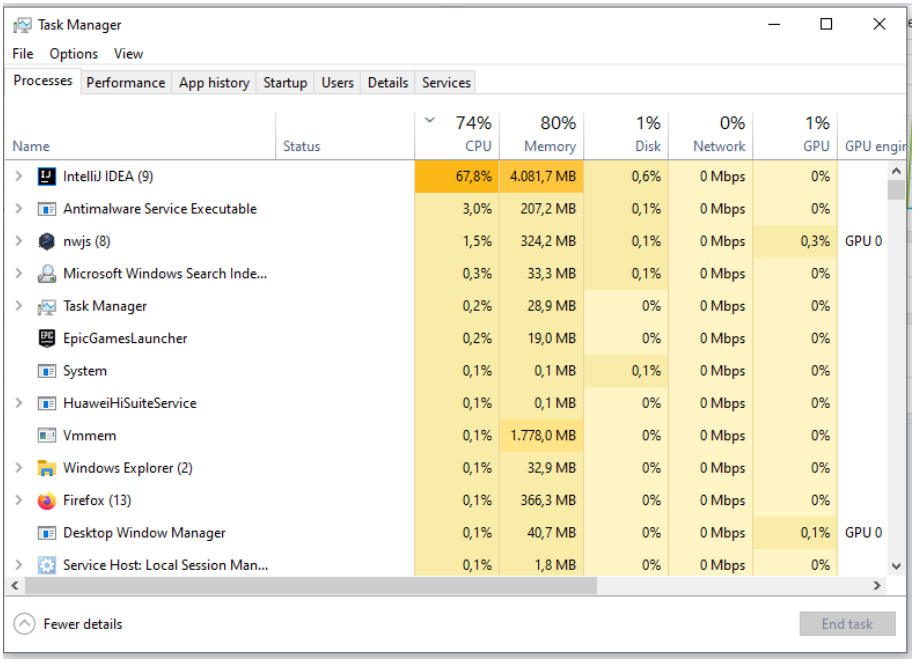


Figure 17: Parallel Execution – CPU and Memory Usage

Some other interesting views concern database sessions and transactions. During the serial execution of the process, active database sessions remain low (Figure 18) and transactions per second are constantly at a high level (Figure 19).

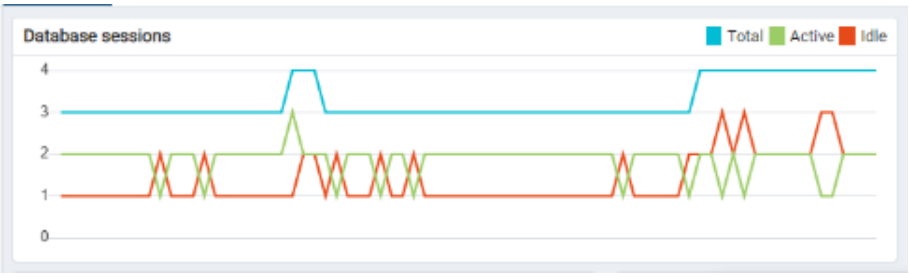


Figure 18: Serial Execution – Active Database Sessions

Figure 19: Serial Execution – Number of transactions per second

In parallel execution, on the other hand, where transactions occur in a batch manner, during the most expensive task, the number of sessions raises momentarily, while the number of transactions remain low.

Figure 20: Parallel Execution – Active Database Sessions

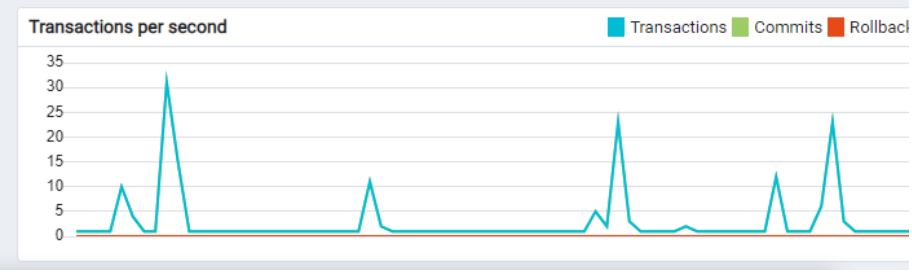


Figure 21: Parallel Execution – Number of transactions per second

6. Conclusions and Future Work

In this work we study the parallelization of Entity Resolution methods, so that they can be applied on Big Data efficiently, while maintaining a high level of effectiveness. Particularly, we focus on blocking and filtering techniques and analyze how to process a record linkage case, where we aim at locating matching entities between two different datasets, using the Improved Suffix Array Blocking approach, enhanced with a Bloom Filter (Vries et al. 2011). We implement this method, both serially and in parallel, in order to compare the results and show whether parallelization makes it suitable for large dataset processing.

6.1 Conclusions

Based on the experiments we conducted, our observations indicate that:

1. The larger the processed dataset is, the greater the difference in performance becomes. In fact, while processing the largest dataset, in our experiments, parallel processing was incredibly (more than 10 times) faster.
2. In case of relatively small datasets, parallel approach is still more efficient than the serial one, but not to the extent we noticed when processing larger datasets. Therefore, selecting parallel methods over a more traditional concept, poses a tradeoff between the profit we have by the performance improvement, over the financial cost of the necessary resources.
3. Parallel execution does not prove to be better than the serial, when applied in very small datasets. In some tasks actually, it shows lower performance.

Regarding Apache Spark, we have also detected some interesting features:

1. Spark achieves an exceptional use of resources concerning CPU and memory utilization. It takes advantage of every resource available to carry out each task and distributes the workload very efficiently.
2. Spark performs computations very fast, especially when there is no need to interact with a database. It justifies in the most emphatic way, the approach of loading input datasets in immutable distributed data structures, and performing computations directly on them.
3. Besides the fact that Spark may not be competitive enough when dealing with smaller datasets, it seems that a threshold regarding data size exists (probably different for each Spark system setup), above which parallelization offers by far better performance.

6.2 Limitations of this study

There were some limitations, regarding the experimental part of this study, mainly concerning the available hardware infrastructures:

- Apache Spark application is executed on local mode because there is no cluster available.
- PostgreSQL Server is also running locally and on the same workstation where the Spark Application is executed.
- It was not possible to use cloud infrastructures (Azure, AWS, etc.)

However, we consider the 12 logical processors CPU, combined with the 16GB of RAM, a decent setup for the purpose of this study.

6.3 Future Work

The purpose of this thesis is to prove that parallelization of ER methods, improves its performance and makes it capable of processing big data. Particularly, in this work we study a specific technique (Improved Suffix Array Blocking with Bloom Filters) which is applied on the blocking stage of ER, only. It would be interesting to include in the experiments the matching stage, also, to have an even better opinion of performance gain while parallelizing the ER process, in total.

We made some useful observations regarding the results of the experiments, and it would be interesting to experiment more on them in order to:

- Fully comprehend the reasons why Spark shows relatively low performance on very small datasets.
- Examine how the parallelized ER performs, with a variation of max block size value.

7. Bibliography

Aizawa, A. & Oyama, K., 2005. *A Fast Linkage Detection Scheme for Multi-Source Information Integration.* [Online]
Available at: http://research.nii.ac.jp/~akiko/papers/wiri2005_aiz.pdf
[Accessed 18 2 2022].

Altowim, Y. & Mehrotra, S., 2017. *Parallel Progressive Approach to Entity Resolution Using MapReduce.* [Online]
Available at: https://ieeexplore.ieee.org/document/7930035
[Accessed 1 11 2020].

Anon., n.d. *https://spark.apache.org/.* [Online].

Benjelloun, O. et al., 2009. Swoosh: a generic approach to entity resolution. *The Vldb Journal,* , 18(1), pp. 255-276.

Bhattacharya, I. & Getoor, L., 2007. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery From Data,* , 1(1), p. 5.

Bloom Burton, H., 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of The ACM.*

Chang, F. W. et al., 2006. *Bigtable: a distributed storage system for structured data.* [Online]
Available at: https://research.google/pubs/pub27898
[Accessed 14 2 2022].

Christen, P., 2008. *Febrl -: an open source data cleaning, deduplication and record linkage system with a graphical user interface.* [Online]
Available at:
http://unstats.un.org/unsd/demographic/meetings/wshops/jordan_21nov10/manuals/febrl_demo.pdf
[Accessed 5 2 2022].

Christen, P., 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9), pp. 1537-1555.

Dean, J. & Ghemawat, S., 2004. *MapReduce: Simplified Data Processing on Large Clusters.* [Online]
Available at: http://research.google.com/archive/mapreduce.html
[Accessed 14 2 2022].

Draisbach, U. & Naumann, F., 2011. *A generalization of blocking and windowing algorithms for duplicate detection.* [Online]
Available at:
http://hpi.de/fileadmin/user_upload/fachgebiete/naumann/publications/2011/icdke2011-a_generalization_of_blocking_and_windowing_algorithms_for_duplicate_detection.pdf
[Accessed 18 2 2022].

Getoor, L. & Machanavajjhala, A., 2012. Entity resolution: theory, practice & open challenges. *Proceedings of The Vldb Endowment,* , 5(12), pp. 2018-2019.

Ghemawat, S., Gobioff, H. & Leung, S.-T., . *The Google File System.* [Online]
Available at: http://research.google.com/archive/gfs.html
[Accessed 14 2 2022].

Jaro, M. A., 1989. Advances in record linkage methodology as applied to the 1985 census of Tampa Florida. *Journal of the American Statistical Association,* , 84(406), p. 414–20.

Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2020. *Learning Spark, 2nd Edition.* s.l.:O'Reilly Media, Inc..

Kane, Frank, 2017. *Frank Kane's Taming Big Data with Apache Spark and Python.* s.l.:Packt Publishing.

Lu, G., 2001. Indexing and Retrieval of Audio: A Survey. *Multimedia Tools and Applications,* , 15(3), pp. 269-290.

Martin Odersky, Lex Spoon, Bill Venners, 2011. *Programming in Scala, 2nd Edition.* s.l.:Artima Inc.

Mauricio A. Hernández, S. J. S., 1995. *The merge/purge problem for large databases.* s.l., s.n.

Papadakis, G., Alexiou, G., Papastefanatos, G. & Koutrika, G., 2015. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. *Proceedings of The Vldb Endowment,* , 9(4), pp. 312-323.

Papadakis, G., Ioannou, E., Niederée, C. & Fankhauser, P., 2011. *Efficient entity resolution for large heterogeneous information spaces.* [Online]
Available at: http://softnet.tuc.gr/~ioannou/publications/files/conf-wsdm-papadakisinf11.pdf
[Accessed 18 2 2022].

Papadakis, G., Skoutas, D., Thanos, E. & Palpanas, T., 2019. A Survey of Blocking and Filtering Techniques for Entity Resolution.. *arXiv: Databases,* , (), p. .

Sunter, A. & Fellegi, I., . A Theory for Record Linkage. *Journal of the American Statistical Association,* , 64(328), p. ''pp.'' 1183–1210.

Vassilis Christophides, et al., 2015. *Entity Resolution in the Web of Data (Synthesis Lectures on the Semantic Web: Theory and Technolog).* s.l.:Morgan & Claypool Publishers.

Vries, T. d., Ke, H., Chawla, S. & Christen, P., 2011. Robust Record Linkage Blocking Using Suffix Arrays and Bloom Filters. *ACM Transactions on Knowledge Discovery From Data,* , 5(2), p. 9.

Wang, J., Kraska, T., Franklin, M. J. & Feng, J., 2012. CrowdER: Crowdsourcing Entity Resolution. *arXiv: Databases,* , (), p. .

Левенштейн, В. И. & Levenshtein, V. I., 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady,* , 10(4), pp. 845–8,707–710.