

Privacy preserving  
record linkage with phonetic algorithms and microservices

Thesis

Konstantinos Razgkelis

dai16250



***ABSTRACT***



The exponential growth of the technology sector and an increasing need for sophisticated structures necessitates a revamping of the current methods. Many organizations are putting a micro services-oriented architecture in place since it is a popular approach for achieving success. In today's environment, a one-size-fits-all approach isn't adequate to provide industries the ability to follow through with creating software with greater durability and demand. Traditionally, we developed projects that started with the concept of expanding always onto a single page and adding layers of new functionalities. When the size of the project has increased and has become enormous, it makes fixing bugs and maintaining security very hard since the mud ball has so many “hard” layers. Therefore, it may hinder the ability of companies to keep up if they do not adopt new frameworks and technologies, since they will have difficulty competing with the increasing demands of the marketplace. This thesis lays out the advantages of microservice architectural design while developing a system that uses many different frameworks to prove the benefits of the microservices oriented architecture. The process of microservices and all the relevant tools and frameworks have all been compiled in the form of containers and images as part of Docker. Docker is a great platform that helps a project team avoid problems when devising and launching any project. Prebuilt images that may be utilized to handle the project's workloads are a critical part of the platform, helping to make project management easy and efficient. The future will provide companies and academics with new materials and tools, and we need to be able to adapt to these new changes to build new systems with the application of modern technology.

***ACKNOWLEDGMENTS***



Before I begin the explanation of this paper, I would like to express my gratitude to Dr. Alexandros Karakasidis, my supervisor, for providing me with the chance to work on this thesis and, with his assistance, to achieve optimum results.

I'd also like to express my gratitude to my family and friends, who have been very supportive throughout my academic years, and I hope that I will be able to demonstrate that they made the right decision to invest their efforts in me.

***CONTENTS***



1. **Introduction**

**2. Related Work**

**3. Problem Formulation and Background**

**4**. ***Methodology***

**5. Empirical Evaluation**

**6. Conclusion and Future Work**

**7. Bibliography and Reference**

***1. INTRODUCTION***



The growth of the technological field has affected all areas of modern life. Companies have modernized their Industry and business plans while people have simplified their daily routines by using all these technological innovations. As technology evolves and the complexity of innovations increases it is only natural that high quality of information will be provided to guide companies taking the best possible actions in an already competitive market. For years, companies have used tools to store data which later are being used to optimize their decisions. Understandably, data usually are numeric values that after a proper analysis offer information which is used for decision making. By that definition, it is safe to assume that data is today’s one of the most important assets that companies, governments and people must consider before making any important decisions.

**“Who has the data has the power”**

Tim O’Reilly 1

The unfortunate reality is that dealing with data is not a simple process. By nature, processing the sheer volume of data that all companies simultaneously produce is a very big burden of a job. It is also inconceivable to transmit them to other hosts while sustaining the protection of this information with other sectors. In most cases, already known sensitive data fields such as names, addresses, and ID numbers are filtered so that they cannot be identified, but other less "essential" variables are not treated equally regarding the protection. Attackers or any harmful individuals may use these fields for their advantage and with specific methodologies, such as joined operation, they can identify people.

As previously said, we have presented the Record Linkage Problem, which is a stumbling block that many data analysts must overcome in their work. Furthermore, with the introduction of GDPR, more stringent restrictions have been implemented to enhance the privacy of data that is sent to third parties who should not be able to identify people. The “Privacy-Preserving Record Linkage” (PPRL) is a term that is frequently used in the literature to express what we have just explained [1]. In solving this issue, however, little attention is given to the fact that the privacy of the specified people in these databases must be protected. When two parties are attempting to discover common records without disclosing any more information to one another, they are referred to as the Privacy-Preserving Record Linkage issue. The introduction of errors, especially from human interaction, can make this problem exponentially harder

Additionally, many organizations as previously mentioned have developed applications that are equipped with tools for carrying out the necessary activities to fully utilize the data that has been produced. Naturally, to remain competitive in the market, every application must be updated regularly in response to market needs. Whether a big or small update is coming, you must have a thorough knowledge of the application's functioning, or else it may cause issues with the system's overall integrity. In certain cases, depending on the size of the program, this may be dangerous since larger applications find it more difficult to adopt newer technologies because it involves cleaning up or changing existing code. If the integration techniques used are accurate and appropriate, the program may be updated without any problems [1]. However, this process is often time-consuming and expensive since it requires a large amount of time and resources. All these problematic attributes belong to the traditional architectural strategy, established in the very early years of application deployment, called monolithic.

We will also discuss solutions to the Privacy-Preserving Linkage problem as well as the constraints of monolithic architecture. The first and most important thing we will do is suggest and explain how phonetic matching algorithms may be utilized to execute operations such as joining two data lists supplied by two separate parties without being restricted by the privacy record linkage. We also offer information on a different design that may be utilized to replace the monolithic architecture, as well as all of the advantages that come along. As a bonus, we will combine these methods into one experiment to demonstrate the advantages of these tactics when correctly executed together in terms of system performance and accuracy of the findings. To put these theoretical models into reality, we will also make use of appropriate contemporary technologies, such as Apache Spark and Docker. Finally, we offer comprehensive examples to allow for a thorough understanding of the intricacies of our approach, and we back up our case for privacy protection with a thorough explanation. Regarding empirical assessment, we conduct comprehensive experiments, to demonstrate the behavior of our approach

***2. Related Work***



The continuous growth of the computational industry generated as a result a very big pool of data. Especially personal data dispersed in multiple data sources, presents enormous opportunities and insights for businesses to explore and leverage the value of linked and integrated data. However, privacy concerns impede sharing or exchanging data for linkage across different organizations. Privacy-preserving record linkage (PPRL) aims to address this problem by identifying and linking records that correspond to the same real-world entity across several data sources held by different parties without revealing any sensitive information about these entities [7]. PPRL is increasingly being required in many real-world application areas .

There have been many examples of applications suggesting the necessity of methods on how to deal with the issue of PPRL. Examples range from public health surveillance to crime and fraud detection and national security. Scalability to multiple large databases, due to their massive volume and the flow of data within Big Data applications, achieving high-quality linkage results in the presence of Big Data's variety and veracity, and maintaining privacy and confidentiality of the entities represented in Big Data collections are all challenges that PPRL for Big Data faces [7].

Relevant work and strong solutions are demonstrated in many researches such as in ‎[8] where they have used cryptographic one-way hash function to conceal the linkage of the information with high accuracy and recall.

Other solutions on how to deal with PPRL are Bloom Filters (BM). BF are space-efficient data structures that have been used for hashing the data while adding protection layers. Unfortunately, this methodology has been proven to be very susceptible to cryptanalysis attacks such as in [11].

There are definitely alternative variations of BF such as in [9] where techniques for securing the BF are demonstrated making BF more resilient to privacy attacks.

We will be offering an overview of strategies that allow businesses to join databases while maintaining data privacy in this study by using the Soundex phonetic algorithm. This method has already been described in [4] and ‎[5] which we will be using with the micro services-oriented architecture to add security layers throughout these organizational communications.

3. ***Problem Formulation and Background***



Detailed descriptions of the issues we are tackling, as well as all the technologies that were utilized to accomplish the desired result, will be provided in this part.

3.1 Monolithic Architecture

In software development, the monolithic architecture is the traditional procedure developers used to create their applications. In terms of deployment, it follows very simple steps regarding its functionality since the system consists of a uniform approach where all of its components are based in a solo system.

In contrast to contemporary designs, this method conveys a straightforward concept of a solid infrastructure that has everything built to its core, with enhancements and updates being processes that get more difficult to complete as the size of the application grows. The most important drawback of this method is that it is compiled by an intriguing number of code lines that are all maintained in a single system. As a result, to introduce new functions or to resolve some of the debugging problems that may arise, an enormous amount of time and human resources would be required.

Despite that, certain unique reasons may make this method particularly helpful especially for tiny applications in their early stages of development. For these “young” applications, monolithic architectures, as stated in [‎1] in “The benefits of the monolithic architecture”, may be advantageous since the development stage can be simpler and less expensive. Because the budget is so restricted in the early phases of any application, the firms who create it typically do not have enough resources to complete the project without using monolithic.

Furthermore, because the code is still in its early stages, making changes to the application is a relatively simple procedure because the code's size is not large enough to complicate things.  Additionally, testing and deployment (the two most essential procedures of any program) may be a simple job, and scalability, such as running numerous instances of the application, is extremely feasible.

It is also important to state that the life cycle of new technologies, from their inception to their "death," was extremely precise. When new technologies enter the market and demonstrate that they are capable of outperforming the previously existing frameworks while also being more cost-effective, they are typically used to replace the "veterans." Unfortunately, the likelihood of this integration occurring in every current application is quite low, because the scope of many applications is generally large enough to prevent such acts from occurring. The concept of rewriting hundreds and millions of lines of code that have been created over a long period to adapt to the new age is just not financially feasible. Integrations are extremely time-consuming, and as a result, there will always be some vulnerabilities in the form of system errors, which will dramatically increase the complexity of the bug repairing process and, thus, the expense of correcting the issue. Furthermore, the sheer scale of the project makes it hard for new individuals to join and contribute to it, since learning the countless code lines will take several months, resulting in a development timeline that is guaranteed to be pushed back significantly.

There are not many applications that remain small as they expand in success and thus, even if monolithic architecture is an excellent approach for delivering the "baby" stages for any program, it is not an appropriate strategy for the application's long-term viability. A different approach necessitates which will be finally introduced.

3.2 Getaway from the monolithic era

In the year 2005, the phrase "Microservices Oriented Architecture" (MOA) was introduced to the world of information technology. From the very beginning, this phrase has called into question many of the individuals who surround the technology sector, calling into question its advantages and, in the end, raising the issue of whether we are ready to move away from the monolithic period or not. Even if there hasn't been a clear victor in the argument over which of the two methods is better since then, the message is clear: "Microservices are here to stay”. Proposing a straightforward, but ultimately gratifying, a solution to the issue discussed above. The idea essentially said that, rather than following the conventional monolithic design, we should develop towards creating applications that are composed of numerous, tiny, and dependable structures, collectively referred to as microservices, rather than a single, large, and unreliable structure. A microservice is a one-of-a-kind and independent component of a larger collection of microservices that together form the application. Each of them has its own set of tasks and duties, and each is capable of functioning independently of the internal state of the other microservices in the system. Furthermore, this design serves as a great future foundation, enabling the program to adjust to new frameworks, expand the size of the application, and perform a variety of other functions.

3.3 Introduction to Microservices-Oriented Architecture (MOA)

As part of the fundamental idea of what makes up a microservice, it is necessary to divide up an application such that all of the separated components present a single microservice entity with specified duties. These entities have their data models, and the administration of their data is accomplished via a particular process that can only be performed inside the confines of the microservice. At the end of the day, each service has its authority and views the rest of the external services as black boxes that can only be accessed via APIs. Even though the microservices by nature are totally isolated, [1] they can still communicate with each other by establishing a link between them via the use of dump pipes or lightweight protocols such as REST API [‎1].

  Let us now see the benefits offered by adopting the micro services-oriented architecture, as outlined in 2.

3.4 MOA: Pros

The significance of each service is immediately apparent since each unique entity is responsible for a particular task and is geographically separated from the others. Microservices are deployable independently of one another, allowing for more team autonomy. In other words, every single one of the microservices should be able to maintain its existence even if the state of all the other services has been adversely impacted by events such as upgrades, debugging, or any other bad occurrence such as bug problems or software malfunction. In the case of a disaster, all instances of the application's microservices may be deployed to various servers in multiple geographical locations, allowing it to be more adaptable than before. Each microservice may be developed and delivered in a variety of geographical locations, including, of course, any location around the globe. Scalability: As the success of an application grows, so do the demands on the application's size. Because each microservice functions as a separate department, each microservice has a unique set of resource requirements. Monolithic applications were used in the past, scaling them up meant scaling up the whole program, which proved to be a very expensive and inefficient approach for most businesses. MOA institutionalizes the resources to every service from the beginning of the application, and when demand grows, the scaling may be focused on the department of interest, allowing for more efficient use of resources. Maintainability and Flexibility: In addition, numerous new frameworks are being developed that may be utilized to keep the application's competitiveness in the market. Historically, this process may be prohibitively costly because it often necessitates a complete restructuring of the app's foundations. The tiny size of the microservice, on the other hand, makes it simpler to integrate it into different frameworks, thus maintaining the application's competitiveness. A smaller codebase makes it easier for the maintainer’s team to grasp the code, enabling new and inexperienced developers to join the project, resulting in quicker development, maintains, flexibility, access and a simpler and more inexpensive to maintain codebase overall. A flawless project, with no minor or significant mistake, could only exist in a utopian or dystopian society. The presence of a bug may hurt the overall state of the application, which will in turn have an impact on the profitability and the client experience. One of the most significant features of this design is its ability to isolate failures inside individual microservices, thus increasing substantially the capacity to detect and repair errors in the system while keeping costs to a minimum. Each service may maintain its integrity without being affected by the performance of the other related entities, as has previously been shown.

3.5 MOA: Possible Disadvantages

Unfortunately, MOA did not appear out of nowhere as a "Deus ex machina" plan to save the world. Neither is it a product that every business must adapt to maintain a competitive position in the market while reaping the advantages of all of its features. There is no such approach, and there should not be one, since it would most likely result in a boring deterministic world, and the need for developers, engineers, and computer scientists would be eliminated. However, this is a philosophical quandary that belongs to a different subject, and the solution should be left to philosophers rather than computer scientists to find. Returning to reality, although this design shines in terms of its ability to deal with the challenges of a monolithic program, it is not without its drawbacks. First and foremost, it is clear that the microservices are separate components of the application's architecture that operate independently of one another. Although, this does not preclude them from cooperating and stressing the necessity of communication among them. As the number of services increases, so does the complexity of the communication system. This necessitates the development of a system capable of handling the large number of requests that must go between the units. In the case of a disaster, if the response time is adversely impacted, a variety of difficulties will arise, and the developers may be compelled to create code to deal with the new challenges, which may include load balancing and network latency, among others. Each microservice has its database, which needs a distinct set of resources to maintain and carry out its functions properly. It is necessary to have experienced developers present to properly integrate from monolithic to MOA architectures. This is often true for the majority of organizations. These trained veterans are priceless assets for any organization, yet they are notoriously tough to obtain. The increasing number of microservices would, without a doubt, raise the need for resources to be distributed correctly, while the right implementation may be a highly complex, time-consuming, and therefore costly process. As the application's size grows and as more services interact with one another, the whole system becomes increasingly susceptible to the difficulties that every distributed system must deal with. When communication complexity grows, the likelihood of a technical issue occurring is substantially increased and debugging issues may be more difficult to deal with. Each microservice has its collection of logs, which may make it a time-consuming process for developers to manually examine all of the files if they are required to do so. Although the advantages of integrating into MOA outweigh any small compilations that may occur, it is essential to highlight the difficulties of fully integrating from a monolithic to a MOA architecture for the sake of impartiality. A "luxurious" approach, in which virtually only applications that have previously been submitted by businesses that can afford the additional cost would reap the benefits. Unfortunately, for new applications that are attempting to quickly enter the market, MOA may be prohibitively expensive in terms of the resources required to do so. As a result, monolithic architecture is much more attractive to freshly developed applications, while MOA is the architecture of the future.

3.6 MOA: Conclusion

To prevent any misunderstanding, it is not suggested that the application has been successfully integrated into MOA as a result of an abstract separation of the application into smaller parts. On the contrary, a thorough study into the construction of the communication system is needed, and the careful design of the foundations is a priority of the utmost importance. Many applications' first stop, right at the beginning of their life, is compulsorily the protective nest of monolithic architecture, where they may ascend in safety, develop quickly, and do so at the lowest possible cost. The third and last stage is the integration into MOA, with the necessity of designing a strategy that will allow the application to develop safely while maintaining the benefits of both methods being emphasized.

3.7 MOA and Monolithic with an example

When describing a monolithic design, you may define it as a black box that is capable of performing the functions of a car. However, the description of a microservice architecture might be defined as a group of black boxes that are capable of accomplishing normal car functions while communicating harmoniously. The first approach is merely focused on the overall outcomes of the entity and pays little attention to the organization and capabilities of its inner components. In the newest rendition, internal parts have significance, emphasizing their distinct functioning, role, and duties, with these all representing the entity under the appropriate collaboration.



*Figure 1: a) At the left is a car presented with Monolithic architecture b) At the right is a car presented with MOA  
source:* [*Microservices Car Example by Fotios Pechlivanis on Dribbble*](https://dribbble.com/shots/16476032-Microservices-Car-Example)

3.8 Enter Docker

The first chapter has shown some of the most often used reasons in support of the use of an MOA-based application design. Based on these assumptions, we will investigate Docker, a piece of software that has many usage capabilities, the most significant of which for this paper is the ability to put the MOA into action in a real-world environment 3. Docker is a framework that helps developers to create projects with very small components which have specific roles. Those components are known as containers and are created by using operating system-level virtualization 4. Besides containers, Docker has many more components that developers use like images, services, networks, volumes, and plugins 3.

3.9 Docker: An Overview

Docker is a free and open platform for building, delivering, and operating apps. Docker allows you to decouple your apps from your infrastructure, allowing you to swiftly release software. You can manage your infrastructure the same way you control your apps with Docker. You may drastically minimize the time between developing code and executing it in production by leveraging Docker's approaches for shipping, testing, and deploying code fast. Docker allows you to bundle and execute a program in a container, which is a loosely separated environment. Because of the isolation and security, you may operate several containers on the same host at the same time. Containers are small and include everything needed to operate an application, so you don't have to rely on what's already on the host. While you're working, you can quickly share containers and ensure that everyone receives the same container that operates the same way 3.

3.10 Docker: Images

A Docker image is a read-only template that contains all of the instructions and commands required to create a Docker container from scratch. Images may be completely modified copies of current images, or they can be fully based on existing images with little customization. Additionally, images may be customized and built from the ground up, although this is often done exclusively by professional and experienced Docker developers. If you want to construct a custom image, you'll need a Dockerfile. This file contains the syntax for describing the essential actions that must be taken to generate and execute the image. It is customary for the Dockerfile to include information on the operating system, programming languages, dependencies, and libraries, among other things. When compared to other competitive virtualization technologies, the most significant advantage of using images is that whenever an update occurs in the instructions of the Dockerfile, only the layers that have not yet been built are created, resulting in a framework that is extremely light-weight, low resource demanding, and extremely fast 4.

3.11 Docker: Containers

Containers are executable instances of one or more images that may be executed at the same time. These instances are separated by using an operating system such as Linux, Windows, or macOS, each of which has its libraries and configuration files.  Containers and virtual machines have a lot in common in terms of functionality. Containers share the services of a single operating system kernel, resulting in their using fewer resources than the most recent version of the operating system. Because the containers are inherently low in weight, an average host system can often handle up to 8 distinct instances at the same time, but this number is not restricted to that particular number in any way 4. Containers may be created, launched, stopped, relocated, and deleted using the Docker API or Docker CLI in a jiffy with the Docker CLI (Command Line Interface). There are many different networks that the containers may be linked to, and each container has its own personal or shared storage system. Last but not least, containers may interact with one another via a variety of internal and external channels that are well specified 5.

Containers have the advantage of being very versatile when it comes to the ways through which they are launched. Even tiny equipment such as personal computers may be a launching station, allowing the development stage to take place while the final product is launched as a temporary or even permanent site at the same location. An application built using Docker is often deployed to a public or private cloud distributed server, such as Amazon Web Services. When speaking to the early phases of development, this demonstrates the limitless potential and amazing scalability of the Docker software, without discounting the final stage of the program at the production level 5.

3.12 Docker: Volumes

By default, when a container is formed, it is given a temporary memory space to store and handle the files or data that is contained inside it. Unfortunately, when the container is terminated, all of the information that was previously stored inside it is then erased as well. Volumes are the most effective method of maintaining and managing the data contained inside a container. They may be a safer alternative to bind mounts when it comes to sharing data across different containers, backing up data, or migrating data. Containers contain a writable layer on top of them, which allows them to store data. Even though volumes are entirely distinct entities and do not belong inside the confines of a container, but rather as external memory space, the size of the volumes may be very dynamic depending on the amount of data stored while retaining the original size of the container host. Last but not least, containers may have several volumes, and vice versa, depending on the configuration 4.

3.13 Docker: Compose

Any of Docker's objects could be built using the Docker API and Docker CLI commands, as previously indicated. Docker has Another essential technology for developers, which allows them to produce more swiftly and simply than ever before and helps creating and organizing a multi-container Docker application. Docker-compose is a docker tool that works with a specific YALM file that stores all of the information needed to create and operate the application. The container's YALM file contains information on container properties such as the container's unique name inside Docker, the IP address, the relationship with linked volumes and networks, and the container's size 4 ,6.

Furthermore, compose may be utilized in a variety of situations, one of which is a development environment, where it can assist in running the program in an isolated and secure environment, among others. Another excellent use is as an automated testing environment, which is an extremely essential element of the process of testing new additions to the application to ensure steady and fast expansion 6. Compose may be used for a variety of activities, and Docker has given developers comprehensive and in-depth instructions on how to utilize it. Lastly, and perhaps most importantly, when compared to the Docker API or CLI commands, the YAML file allows for more straightforward persistence of the configuration files while keeping them in a readable format, which significantly simplifies the developer's job to build without making the update procedure overly complicated 4.

3.14 Docker: Docker Hub and repositories

Finally, the enormous number of private and public images available in Docker's repository makes it one of the finest platforms for developing MOA applications. It contains more than 100,000 distinct pictures, each of which has been downloaded more than a billion times. Some of the pre-built images that were utilized for this research include the PySpark-notebook from Jupyter, Hadoop, Apache Spark, and Ubuntu, to name a few 4.

3.15 Docker: The final word

During the same time that MOA's popularity is increasing, so is the need for a software connection that can offer a steady interface between theory and practice. Docker's extensive documentation, straightforward implementation, and scalability make it a valuable tool for developers who are transitioning from a monolithic to a modular architecture.

The introduction of our first problem, which was related to the architecture of our application, has now been completed, and we have also given a solution. As a result, it is past time for us to begin discussing the second problem in detail.

3.16 Privacy-Preserving Record Linkage (PPRL)

The following mathematical formula, theory, and example are used to explain the problem we are trying to cope with and it is taken from [4].

Let us consider two data sources, called Alice (A) and Bob (B), who respectively hold and records each. We denote as and the i-th record of Alice and Bob, respectively. We represent the j-th attribute of these records as .j and .j.

Privacy-preserving record linkage is the problem of identifying (linking) all pairs of and records that refer to the same real world entity, so that no more information is disclosed to either A, B , or any third party involved in the process besides the identifiers of the linked s and [‎4].

Alice and Bob will most probably use different schemas in their databases. As such, they will have different attributes. Let be Alice’s schema and be Bob’s schema and let us assume that in these schemas m of the attributes is common between the two sources forming a composite key. These attributes might be names, surnames, addresses, and birth dates. As such, none of these on its own may comprise a unique identifier that can be used to identify a record. We refer to these attributes as matching attributes or matching fields. The composite key is used to determine when two records match, i.e., when they refer to the same entity. To determine when two records match, the respective attributes forming the composite key need to be compared. Considering that our data is often dirty, matching should rely on a similarity or distance function [7].

Let us consider a similarity function () → [0...1] and a threshold tj > 0. Given the records and with matching attributes .1 ....m for both Alice and Bob, we define the following matching function M → {0, 1}:

[1]

If M ( , ) = 1, then the pair ( , ) is a match.

This process is the matching process. To preserve privacy, i.e., ensure privacy-preserving matching (PPM), after the completion of this process, the only information revealed is the identifiers of the matched records [3].

3.17 Phonetic Algorithm: Introduction

A phonetic algorithm is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for [indexing](https://en.wikipedia.org/wiki/Index_(publishing)) of [words](https://en.wikipedia.org/wiki/Word) by their [pronunciation](https://en.wikipedia.org/wiki/Pronunciation) 7 . It's been well over a century since the original algorithm Soundex was created [4] to help with name matching tasks, and since then, similar techniques have been used often for name-based record matching, with recent years seeing a huge uptick in their popularity Most of the terms discovered in it are contained in the English language. For the sake of this study, we shall narrow our attention to Soundex, which was utilized in the experimental work. Many people know that phonetic algorithms are very resilient in the face of typing mistakes and can operate even when broken [‎4].

3.18 Phonetic Algorithm: Soundex

Soundex [6] is the first phonetic algorithm that is responsible for establishing a connection between words by producing a coding pattern for them based on their similarities. This is how it is capable of creating a link between them. The name's initial letter was used for the first letter of the code, while the three phonetic sounds of the name provided the three letters of the last three letters of the code. The Soundex code helps differentiate homophone words since it uses a series of numbers to identify each instance of a homophone word. These homophone words are separate, as they sound alike but have different spellings. Additionally, the output will contain a single letter and three numbers, each of which has been changed by the algorithm.

3.19 Phonetic Algorithm: Example

Soundex may be better explained by using the example of how it handles the phrase "The amazing Spiderman". Soundex's transition follows a strict routine and process; it is not just an unorganized change. The first digit of the Soundex-corrected transformation begins with the same letter as the first letter of the word being converted. So, left vowels are discarded in addition to "y," "h," and "w". These next three numbers are as follows, in the pattern laid out in the table shown below, and are defined using both significant and minor words, with the use of 0s to fill in for small, irrelevant words. Using the Soundex method, the sentence of the example might be transformed into "T000 A525 S136."

| Letters | Soundex |
| --- | --- |
| a, e, i, o, u, y, h, w | 0 |
| b, f, p, v | 1 |
| c, g, j, k, q, s, x, z | 2 |
| d, t | 3 |
| l | 4 |
| m, n | 5 |
| r | 6 |

*Table 1*

3.20 Phonetic Algorithm: Privacy-Preserving Record Linkage Using Phonetic Codes

It is important to keep your information confidential. Identifying all record pairs of the same real-world entity in a way that A or B or other third parties participating in the process will not receive any additional information, except the identification of associated record pairs, is known as a Privacy-Preserving Record Linkage [‎5]. We have previously identified Soundex as a tool that can be used to solve this specific problem, and it is capable of performing the matching procedure in data.

To comprehend how PPRL with Soundex works, we will be using the same example used in the paper [‎4] we must first bring the well-known Alice, Bob, and Carol into our environment. Alice and Bob will represent the parties whose data will be impacted by PPRL, with Carol performing the joint action later on. Before submitting their data to Carol, Alice and Bob must complete a certain number of steps. We will be using the table with the algorithm from [5].

1. Their data is changed using a Map function to be converted to the Soundex output produced by the function

2. Create a data list that contains noise by utilizing fictitious values derived from random Soundex findings.

3. Combine the two data sets into a single one.

4. Apply a hash function to the new dataset to filter it.

5. Sort the hashed dataset into a random order

6. Send the information to Carol.

Now Carol has both data files and can start the join action to them. Finally, the joined data will be sent back to Alice and Bob.

3.21 Apache Spark: Introduction

Apache Spark is a free and open-source engine for large-scale applications involving enormous quantities of data to be processed. Almost all well-known online platforms, as well as the vast majority of big companies, rapidly embraced Spark from the outset. Spark is mostly used in the data analytics and machine learning sectors because of its ability to handle massive quantities of data. Spark is a descendant of Apache Hadoop, and it is capable of supporting a wide range of programming languages and frameworks. It takes the place of the previous program by doing the same function in less time 4, 7.

3.22 Apache Spark: Foundation

Spark Core serves as the platform's basis. These duties include memory management, fault recovery, scheduling, distributing, and monitoring jobs, as well as interacting with storage systems, and they are all handled by this component. Spark Core is accessed via an application programming interface (API) that is available for Java, Scala, Python, and R. Simple, high-level operators in these APIs conceal the complexity of distributed processing, which makes it easier to use 8 .

MLlib, a library of methods for large-scale machine learning, is included in the Spark framework. Training Machine Learning models using R or Python on any Hadoop data source, saving them with MLlib, and importing them into a Java or Scala-based pipeline are all options available to data scientists. Spark was developed to facilitate machine learning by providing a rapid, interactive computing environment that works entirely in memory. Among the methods accessible are classification, regression, clustering, collaborative filtering, and pattern mining, to name a few examples 8.

Because of the rapid scheduling capabilities of Spark Core, Spark Streaming is an ideal real-time streaming analytics solution for large data sets. It ingests data in mini-batches and conducts analytics on the data using the same batch analytics application code that was used for the previous batch 8. It is possible to use the same code for batch processing and real-time streaming applications, which improves the productivity of the developer 8.

If you want low-latency, interactive queries, Spark SQL is 100 times faster than MapReduce when it comes to delivering that service. While scaling to hundreds of nodes, it includes a cost-based optimizer, columnar storage, and code generation for rapid queries to ensure that the system remains fast and responsive. Business analysts may query data using either conventional SQL or the Hive Query Language, depending on their needs 8.

Spark GraphX is a distributed graph processing system built on the Spark platform. With the use of ETL, exploratory analysis, and iterative graph computing, users may interactively create and modify a graph data structure on a large scale. In addition to having a very flexible API, it also includes a variety of distributed graph algorithms built in 8.

3.23 Spark: Resilient Distributed Datasets (RDDs)

In computing, a resilient distributed dataset (RDD) is a read-only collection of objects that have been partitioned over many computers and that can be recreated if a partition has been lost [‎9]. RDD components do not need to be physically kept; instead, an RDD handle has enough information to create an RDD from data obtained from a trustworthy source. Even if a node fails, RDDs can always be rebuilt, ensuring that the system remains operational.

3.24 Apache Spark: Apache Spark over similar frameworks

Obviously, there have been many other frameworks that existed before the creation of Spark and to demonstrate the differences and significance of spark we should make a simple comparison.

Apache Hadoop is one of these tools that was used before Spark. Hadoop utilizes the MapReduce algorithm to process large data sets. In the case of massively parallelized operators, developers are relieved of the burden of worrying about task allocation and fault tolerance issues. MapReduce, on the other hand, has difficulty dealing with the sequential multi-step procedure required to complete a job. Towards the conclusion of each phase, MapReduce gets data from the cluster, executes operations on it, and publishes the results back to HDFS, where they belong. Because each step in a MapReduce job requires a disk read and write, MapReduce workloads are much slower than traditional computing tasks 8.

By performing processing in memory, decreasing the number of steps required to complete a task, and reusing data across many concurrent processes, Spark was designed to overcome the constraints of the MapReduce framework. Because Spark receives data into memory in a single step, performs operations, and sends out the results, it is much quicker than other programming languages. Spark also reuses data by storing it in an in-memory cache, which allows machine learning algorithms that repeatedly execute the same function on the same dataset to perform much faster.

Reusability of data is enabled through the creation of DataFrames, which are an abstraction over the Resilient Distributed Dataset (RDD). When conducting machine learning and interactive analytics, this lowers latency significantly, resulting in Spark being several times quicker than MapReduce.

3.25 Apache Spark: lazy evaluation

Lazy evaluation, also known as call-by-need in programming language theory, is an evaluation approach that avoids repetitive evaluations by deferring the evaluation of an expression until its value is required. It is a programming language technique that is used to evaluate expressions when their values are required (non-strict evaluation). The use of other, non-strict evaluation techniques, such as call-by-name, which evaluates the same function again and blindly, regardless of whether or not the function may be memoized, can reduce the execution time of some functions by an exponential factor, depending on the function 10.

Computer systems may be made faster by storing the results of costly function calls and returning the cached result when the same inputs are used again. Memoization, also known as memoisation, is a method for speeding up computer systems. As an example, basic mutually recursive descent parsing, which is used for purposes other than speed, has made use of memory in various contexts. Even though it is linked to caching, memoization refers to a particular instance of this optimization, as opposed to buffering and page replacement, which are both types of caching in themselves. In several logic programming languages, the term "Memoization'' refers to the act of postponing something. By using this programming and processing approach, the pace of the spark is considerably improved, allowing it to do the computational jobs much more quickly while decreasing the likelihood of a mistake 11.

3.26 ETL pipeline

Extract, transform, and load (ETL) is a three-phase computing process that involves extracting data, then transforming it (cleaning, sanitizing, and scrubbing it), and lastly loading it into an output data container. Data may be collected from a variety of sources and outputted to a variety of destinations. ETL processing is usually done with software tools, although it may also be done by system operators manually. ETL software often automates the whole process and may be performed manually or on a recurring schedule as individual tasks or as a batch of jobs 11.

An ETL system that is correctly built takes data from source systems, enforces data type and data validity criteria, and guarantees that it complies structurally with the output requirements. Some ETL systems can also produce data in a presentation-ready manner, allowing application developers and end-users to make informed decisions 12.

In the 1970s, the ETL process became a popular idea, and it is now widely employed in data warehousing. ETL systems frequently combine data from a variety of applications (systems), which are generally produced and supported by various vendors or housed on different computer hardware. Different stakeholders commonly maintain and administer the distinct systems containing the original data. A cost accounting system, for example, may incorporate data from payroll, sales, and buying 12.

Recently, we've covered a wide range of tools and frameworks, their capabilities, advantages, and disadvantages, as well as the reasons why they've been employed in certain situations. In addition, we demonstrated the significance of privacy-preserving record linkage, the fundamental idea of micro services-oriented architecture, and several tools that may be used to put these concepts into practice. Finally, in this chapter, we will describe how all of these frameworks may be linked together in such a manner that an ecosystem can be built to assist us in doing PPRL utilizing Microservices. Before we start with the explanation of our system, we need to announce that the experimental part was concluded in two versions. The initial version was built with a fully functional solution, which was typically accomplished by using some of the fundamental commands of each framework. An incomplete solution using complex instruments and methods is designated as the second version, with the hope of improving our experiment's outcome in the future. We will begin with the basic solution and later on, we will go through the improved implementation and any future updates that may be necessary.

4. ***Methodology***

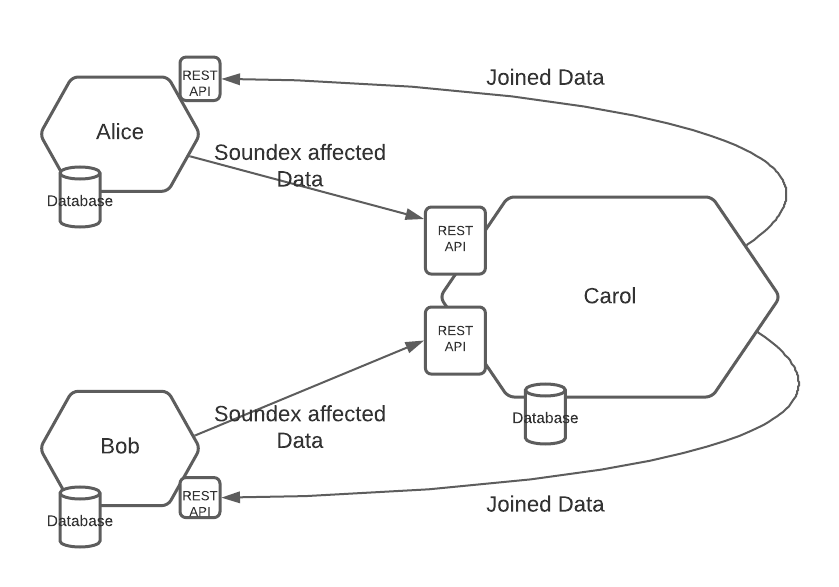


In this part, we provide the results of our experimental assessment of our method. In this study, we will show the processing time to complete the tasks using the micro services-oriented architecture on our datasets.

4.1 Experimental Setup

We will first need to give a demonstration of the hardware that was used in order to perform our operations. We ran our solution via the Okeanos IaaS academic service to ensure they worked properly. Our MOA system will be composed of three microservices that will operate as Docker containers and from which we will get the data that has been stored in their storage. Also, we will do the same experiment with noise ranging from 0 to 1000 percent of the total data amount. Our interface will be Jupyter Notebook, which we will use for both. With a 4-core processor and 8 GB of RAM, the virtual machine (VM) is fully functional.

First and foremost, we must address our microservices and the responsibilities they bear. We will create a basic system in which there will be just two entities with data sources and one entity that will link them and produce the necessary information back to them. As a result, for the time being, we may begin our experiment by creating three microservices: Alice, Bob, and Carol. Alice and Bob will serve as the two entities with their essential data, while Carol will operate as the mediator, bringing their data together in a safe and secured environment. Figure 3 depicts the design pattern that we used to solve our issue.



*Figure 3*

4.2 Introduction of the classes

Before we start introducing our services, we will show the custom functions that were used to apply the Soundex algorithm, SHA algorithm, and the methods that create the fake noise. Inside the container of Alice and Bob, in the directory ‘packages’ there are two important files, the **etl\_pipeline.py,** and the **transformation\_functions.py**.

4.3 Custom transformation functions

The transformation functions are being used in the ETL pipeline to create fake Soundex values. Therefore, we will briefly introduce the methods of this class. Below, there is the table with the functions and their description.

| function | description |
| --- | --- |
| **create\_alp()** | is the first function and when is called it generates a random Soundex value (e.g. N310, B902). |
| **create\_fake\_soundex\_values(noise, dataframe, m\_field):** | This function when called takes a dataframe and regarding the percentage of the noise, it returns a dataframe with fake values. It also creates a different column for the matching field which is not being transformed |
| **create\_noise(noise, size, name)** | This function creates a series of added noise and returns it. |
| **create\_fake\_index(noise, size , name)** | Creates a column field with the fake index as a value |



| **import** random **import** pandas as pd   def **create\_alp**() -> str:   """ This functions is beings used **for** the creation of fake  soundex entries    Returns:  str: a Soundex entry e.g. N310, B902 ...  """    soundex\_value = str(chr(random.randrange(65,90)))\  + str(chr(random.randrange(48,54)))\  + str(chr(random.randrange(48,54)))\  + str(chr(random.randrange(48,54)))    **return** soundex\_value |
| --- |

*Code 1*

| def **create\_fake\_soundex\_values**( noise : **int**,  dataframe : pd.DataFrame,  m\_field : str) -> pd.DataFrame:    """ Creates a dataframe with a Soundex values which will be merged  with our dataset in order to add layers of noise. This will  apply extra protection to our data.    Args:  **noise** (**int**): The percentage of the noise we are going to use in  our **dataset** (e.g. 10%, 20%)   **dataframe** (pd.DataFrame): The dataframe with the data  **m\_field** (str): the column which will be used to make the join  operation to find our matches    Returns:  pd.DataFrame: The dataset with the fake soundex values  """  size = dataframe.shape[0]    fake\_index = create\_fake\_index(noise, size, m\_field)  df = pd.DataFrame(fake\_index, columns=[m\_field])    **for** row in dataframe.columns:  **if** row != m\_field:  noise\_df = pd.Series(create\_noise(noise, size, row), name=row)  df = pd.concat([df, noise\_df], axis=1)    **return** df |
| --- |

*Code 2*

| def **create\_noise**(noise : **int**, size : **int**, name : str) -> pd.Series:    """ This function is being used to add noise in our dataset.  It **requires** the percentage of the **noise** (e.g. 10%, 20%)  the size of the **dataset** (e.g 10.000 entries) and the name  of the column.    Args:  **noise** (**int**): The percentage of the noise we are going to use  in our **dataset** (e.g. 10%, 20%)  **size** (**int**): The size of the **dataset** (e.g. 25.000, 50.000)  **name** (str): The name of the column    Returns:  pd.Series: A series with the soundex values values  """    se = pd.Series([create\_alp() **for** \_ in **range**(**int**(noise \* size / 100))]\  , dtype = 'string'  , name=name)  **return** se |
| --- |

*Code 3*

| def **create\_fake\_index**(noise : **int**, size : **int**, name : str) -> pd.Series:  """Creates a column field with the fake index as a value    Args:  **noise** (**int**): The percentage of the noise we are going to use  in our **dataset** (e.g. 10%, 20%)  **size** (**int**): The size of the **dataset** (e.g. 25.000, 50.000)  **name** (str): The name of the column    Returns:  pd.Series: \_description\_  """  se = pd.Series(['Fake Index' **for** \_ in **range**(**int**(noise \* size / 100))]\  , dtype = 'string'  , name=name)    **return** se |
| --- |

*Code 4*

The explanation of the transformation functions is done, and we will continue the explanation of the ETL pipeline

4.4 ETL pipeline

Since the idea behind our task is to create a web application, we should be using a proper method to Extract, Transform and Load the data. The **etl\_pipeline.py** is a custom class that provides that specific ability to our application. The ETL pipeline, as already mentioned, is being used in both Alice and Bob since they require a proper architecture to properly process the data and later send them to Carol. Therefore, we need to make a deeper explanation of this custom-made class and the capabilities of all its functions.

| **import** pandas as pd **import** packages.transformation\_functions as tf **import** jellyfish **import** hashlib   PATH = f'/var/lib/data/datafile.csv'   **class** **ETLModel**:  """  **This** **is** **the** **ETL** **class** **model** **that** **will** **be** **used** **in the backend** **for** **our** **application**.  **The** **ETL** **pipeline** **is** **responsible** **for** **the** **extraction**, **transformation** **and** **loading**  **the** **data**.  """  **def** **\_\_init\_\_**(**self**, **matching\_field** : **str**, **noise** : **int**):  **self**.**matching\_field** = matching\_field  self.noise = noise  self.dataframe = None |
| --- |

*Code 5*

For the initialization of the ***ETLModel*** object, it is required to provide the arguments ***matching\_field*** and ***noise***. The ***matching\_field*** is the field of our data frame that will be used to identify the possible matches and thus it should not be changed with the Soundex or the hash functions. The ***noise*** is the percentage of the fake Soundex values in our data frame. Finally, the object initializes the dataframe variable which is being used throughout the pipeline. The ***PATH*** is a global variable that gives the path inside the container where our file is stored. We will be using the datafile.csv in our ETL pipeline.

Usually, well-established and completed applications take data from sources using API calls. Since it would require a lot of time to set up a database to store the data and an API to call them, we will be using the easiest and faster solution of file dumping to move the data around the containers.

The first function in our **ETLModel** class is the **extract\_data**. This function reads a csv file from the container’s storage (found in the PATH provided) and creates a pandas dataframe.

| def **extract\_data**(self) -> pd.DataFrame:  """ This function extracts the data from a source location.  (e.g csv file)    Returns:  pd.DataFrame: A pandas dataframe  """  self.dataframe = pd.read\_csv(correct\_path, header=0)\  .astype('string') |
| --- |

*Code 6*

The transform data method is the second function in our **ETLModel** class. This function starts altering the dataframe that was produced by the **extract\_data** function. There are several stages to the transformation process.

**Stage 1**: We must first create the **columns\_we\_care** list, which contains the fields that will be transformed using the Soundex and SHA functions. All of the dataframe's fields will be added, except for the **matching\_field**, which will be removed.

**Stage 2**: Now that the fields have been divided, we'll begin adding bogus Soundex values based on the **noise** value. We'll get a dataframe with fake Soundex values and a series with fake values for the matching field if we use the **create\_fake\_soundex\_ values** function from the **transformation\_functions** library. It's worth noting that the SHA function is also used with the dataframe and series.

**Stage 3:** We will apply the Soundex and the SHA to real data in this stage.

**Stage 4**: We'll now combine the false hashed Soundex values with the data's hashed Soundex values. We'll sort the combined data set by one of the fields (we'll use field = 'the first name,' but any of the other values will suffice).

**Stage 5**: Finally, we'll remove the automatically formed index, which may otherwise cause issues because it could be utilized as an identity field.



| def **transform\_data**(self) -> pd.DataFrame:  """ This function take the extracted dataframe and starts  transforming the **data** (cleans the data)    Args:  **matching\_field** (str): the column which will be used to  make the join operation to find  our matches  **noise** (**int**): The percentage of the noise we are going  to use in our **dataset** (e.g. 10%, 20%)   **dataframe** (pd.DataFrame): The dataframe with the data    Returns:  pd.DataFrame: The transformed data  """    columns\_we\_care = ['NCID', 'last\_name', 'first\_name', 'midl\_name',  'street\_name', 'res\_city\_desc', 'birth\_age']    columns\_we\_care.remove(self.matching\_field)    fake\_soundex\_values = tf.create\_fake\_soundex\_values(self.noise,  self.dataframe,  self.matching\_field)    **for** column in columns\_we\_care:  #Applies to the data the jellyfish-soundex function  #and also encodes them with the SHA256 encryption  self.dataframe[column] = self.dataframe[column]\  .apply(lambda x: jellyfish.soundex(str(x)))\  .apply(lambda x: hashlib.sha256( x.encode())\  .hexdigest() )    #Creates fake soundex values  fake\_soundex\_values[column] = fake\_soundex\_values[column]\  .apply(lambda x: hashlib.sha256( x.encode())\  .hexdigest() )    #We merge all the fake soundex data with the correct  merged\_data = pd.concat([self.dataframe, fake\_soundex\_values], axis=0)\  .sort\_values(by='first\_name')    merged\_data.reset\_index(drop=True, inplace=True)    self.dataframe = merged\_data |
| --- |
|  |

*Code 7*

The third function in the **ETLModel** is the **load\_data** whichsimply stores the transformed dataframe in the storage. In our application, it will be stored as a dump file in a CSV format.

| def **load\_data**(self):  """ Loads the data to a **dumpfile** (csv file) and stores them in the storage inside the container    Args:  **dataframe** (pd.DataFrame): \_description\_  """  self.dataframe.**to\_csv**('/var/lib/data/hidden\_information.csv',  encoding='utf-8', header=True, index=False) |
| --- |

*Code 8*

The fourth and final function in the **ETLModel** is the **start\_etl** which triggers the start of the pipeline. The trigger has occurred in the functions of our API endpoints which will be discussed later.

| **def start\_etl(self):  """  Start the etl pipeline  """  self.extract\_data()  self.transform\_data()  self.load\_data()** |
| --- |

*Code 9*

4.5 Alice/Bob service setup

Docker is the primary program with which we will integrate all our technologies to form a cohesive whole. We will be using the docker-compose capability, and as a result, we will need a more in-depth knowledge of the setup of the “docker-compose.yml” file to proceed. This file contains an explanation of all of the configuration options that have been applied to the microservices. In code 4, we can see the “docker-compose.yml” file that has been defined, and we will go through each of its characteristics one at a time as follows 13:

| **In the section "version" we define the version of the .yml file since different version requires different commands for the proper configuration   version: '3.0'   In the section "services" we define the microservices of our problem and we place all the need requirements or settings   services:  cluster-a:  container\_name: cluster-a  build:  context: ./services/data-service-1  dockerfile: Dockerfile  ports:  - "9200:9200"  environment:  - PORT=9200  volumes:  - cluster-A-volume:/var/lib/data    cluster-b:  container\_name: cluster-b  build:  context: ./services/data-service-2  dockerfile: Dockerfile  ports:  - "9300:9300"  environment:  - PORT=9300  volumes:  - cluster-B-volume:/var/lib/data    jupyter:  container\_name: jupyter  build:  context: ./services/jupyter  dockerfile: Dockerfile  ports:  - "8888:8888"  volumes:  - shared-workspace:/opt/workspace  networks:  - spark-net    In the section "volumes " we define the volumes that we will need for our microservices. In this place only the objects are created and no further connection with the services is configured. The connection with the services is established to the services configuration.   volumes:   cluster-A-volume:  cluster-B-volume:  shared-workspace:  name: "hadoop-distributed-file-system"  driver: local   networks:  spark-net:** |
| --- |

*Code 10. This is the docker-compose.yml files. 1*

‘‘Version" is the first command that we have to establish. Version means defining the version of the docker-compose.yml files. There are three constrained versions, (Version 1, Version 2, and Version 3) which each have unique specifications. In this experiment, we will be referring to Version 3 since we need to implement the capabilities of all versions.

Under the “services” area, we will begin configuring our microservices and all the parameters that we want to add. As can be readily observed, all the services have almost similar configurations, with a few minor variations that are typically determined by the duties of each service.

To properly develop the experiment, it is essential to refer to each container by its unique identification character. Fortunately, the command “container name” makes this feasible by storing a String containing the name of the container, as stated. This would be a fantastic idea later on when we are faced with the debugging process since we could isolate the issues by examining the log files for the name of the container.

Self-explanatory, the command "build" is building the container adding all the dependencies, files, and modules. It will begin creating the container from either an image fetched from the Docker Hub repository, or from the information provided in a particular file known as the Dockerfile, depending on the situation. In our example, all of the services will be built based on a custom Dockerfile, which will be described in detail later in the paper with extensive details. Lastly, all configuration files generated during the container's construction are stored in a particular place determined by the information provided in the "context" portion of the container's construction. The term "context" simply refers to the place where the container will be kept. It is only when the container is being built based on the customized Dockerfile that the Dockerfile instructions are shown. Otherwise, this part is removed and replaced with an image section, which specifies the location of an image on which our container will be built as a foundation.

Services are instances of the application that run independently of the rest of the program. We must first create a link between the services for them to be able to use the functionality of the other services. As a result, we must specify the ports on which all of the microservices will listen for communication. This is easily accomplished via the use of the section "PORT", which allows us to declare the port number of each microservice. Given that certain microservices may use more than one port, we may simply provide several port numbers in this section to accommodate this possibility. It is also essential to emphasize that the port numbers for both services should be always different since one service would override the functionality of the other if the port numbers are not the same.

Some of the configuration instructions may also be registered in the “environment” section of the configuration file. This section typically contains a list of the most essential variables that a container must have before it can be launched. This information may include account credentials (username and password), the size of the container, and the user's rights inside the Linux kernel, among other things. As shown in Figure 2, we may as well specify the port number by including it in the "environment" portion of the code.

Containers may also be implanted with shared or private storage, depending on the configuration. In the section "volume," we may link the container to various existing volumes by providing the volume names in this section.

The Volumes portion of the compose-file is the last and most significant element. As shown in the code's last lines, Volumes may be generated quickly and easily by just declaring their names. Several other volume settings might be used in conjunction with a given issue. However, we will not include any of these unique options in the initial iteration of our experiment.

We have finally presented our compose-file and discussed all of the settings that have been applied to our microservices, and we are nearly through with the preparations for launching our system. We must first explain the Dockerfile for this to take place. Dockerfile is yet another one-of-a-kind configuration file that we must learn more about before proceeding with our example.

4.6 Dockerfile-Alice and Bob

As described later, we said that every container could be built based on a prebuilt image obtained from the Docker Hub repository, or it could be built from scratch using a unique design. Additionally, a hybrid option is available, which involves modifying an already-built image, which is often the most popular and efficient approach. To do this, we have created a Dockerfile, in which we will describe in detail the commands that are used to configure it.

First, we need to set up our environment requirements which consist of all the modules and packages our application will be using.

The requirements.txt for all containers:

| Name | Version |
| --- | --- |
| Flask | 1.1.4 |
| requests | 2.23.0 |
| Markup safe | 2.0.1 |
| Jellyfish | 0.8.2 |
| Wget | 3.2 |
| pandas | 1.3.4 |

In the sense that Alice and Bob are two comparable entities (that is, two microservices that provide the same task), their Dockerfile is virtually identical by nature. This implies we will only have to go through one of these two Dockerfiles since they both express the same functionality, which is a relief. Alternatively, Carol has a distinct set of capabilities, and although the Dockerfile for Carol may have some similarities with the ones used by Alice and Bob, there are certain differences that we will need to consider as well. In Code 11, we can see Alice and Bob Dockerfile 13.

| Defines the image of which the container is going to be based FROM python:3.9   This is the directory where our system will work on WORKDIR /var/lib   The containers' copies in its storage the requirements.txt file from the local machine This file is filled with all necessary libraries that we need to install before the container's launch   COPY requirements.txt ./   Here we run the pip commands which installs the libraries in our Linux Kernel RUN pip install -r requirements.txt   It copies all the files in this specific location COPY . /var/lib/   It defines the port number the container should expose EXPOSE 9200   Finally, in order our container to run the python code we do it with final command CMD [ "python", "./app.py" ] |
| --- |

*Code 11 This is the Docker file of Alice and Bob* 13

Using the command "FROM," we begin the process of defining how our container will be built, namely whether it will be built from an image or if it will be built from scratch. When building the container for Alice and Bob, the image python will be used as a starting point, which is an instance of a computer running on the Ubuntu operating system with python3 already installed. Consequently, we have already specified the operating system and programming language with a single simple and quick operation.

It is now necessary to provide the working directory. Simply put, the command "WORKDIR" instructs the operating system to proceed to the place where it will be doing its duties. In this case, we are directing the operating system's attention to the directory /var/lib. Therefore, all of the following activities will be carried out in that specific location alone.

The container's primary responsibility is to execute the commands specified in the main code source; therefore, it is essential to implement all the required dependencies and libraries. This is accomplished by copying all this information from the local host computer to the container operating system in a file named "requirements.txt" that is deployed. A simple way to do this is to use the command "COPY," which should be followed by the file name and, lastly, the location to be copied or saved, as shown. Additionally, the requirements.txt file is mostly used for the importing of modules/libraries that are compatible with a particular version of the framework.

Now that we have a file with the necessary information about the libraries, we just need to install them on our container. We may begin our installation by entering the command "RUN" followed by the appropriate operating system instructions, such as pip (which is the command to install python libraries) or sudo, among others.

In the next step, we must use the command "COPY" once again to copy any additional files that may be required, as well as specify the proper port number to be used as a gate listener. The port with the number 9200 will be exposed in our Alice container.

Finally, and perhaps most crucially, we must execute our main code for the container to do its duties properly. Through the use of the command "CMD," the computer is initiating a command panel activity. Simple inference can be drawn from the terms “python” and “./app.py,” which indicate that the operating system will execute a python script with the name app.py that has been stored in the specified directory.

According to the above statement, Alice and Bob have the same characteristics, which means that their Dockerfiles have almost identical implementations. The only difference is that Bob is utilizing a different port number, 9300, to expose its listening gateway when compared to the other. As a result, there is no sense in explaining the Dockerfile of Bob in full again since the instructions are essentially the same. Carol's container, on the other hand, is built using a different method, and it will be very interesting to examine the Dockerfile instructions that are used to create them.

4.7 Dockerfile-Carol

| For our interface, we will be using the jupyter The notebook helps us speed our processing.   FROM jupyter/all-spark-notebook:spark-3.2.1   It is an excellent practice to add users to our container so that we can define different roles in our web service. We will be using only root for this example   USER root   We also need to copy in the container the requirements.txt and SparkCommands.ipynb files. The first one will have all the modules and packages that the service will    COPY requirements.txt ./ COPY SparkCommands.ipynb ./   Here we run the pip commands which install the libraries in our Linux Kernel   RUN pip3 install -r requirements.txt   It defines the port number the container should expose EXPOSE 8888   Finally, in order for our container to run the python code we do it with a final command   CMD ["jupyter", "notebook", "--port=8888", "--no-browser",  "--ip=0.0.0.0", "--allow-root"] |
| --- |

*Code 12. This is the Dockerfile for Carol.* 13

We must utilize an operating system on which our container will be built, just as Alice and Bob did. For this container, we will be using instead an official Docker Hub image 14.

Our second step is to specify the user and his or her permissions in our operating system as simply as possible by using the “USER” command on the command line. That is a straightforward process, and we now have a user with appropriate permissions in our system.

We will need to implement the dependencies and libraries from the particular requirements.txt file once again using the “COPY” command. We will start installing the dependencies required in our requirements.txt file. We will also expose the port 8888 for our container to be able to listen to any external requests. The container will start by using the “CMD” command at the end of this process.

Following a successful presentation of our containers' Dockerfiles, we are now prepared to describe the functioning of each container in more detail. As a result, we will present the primary source code for each container in this section.

4.8 Main Code – Alice and Bob

In the final line of their Dockerfiles, we have instructed our container to start a particular file in Python using the command line arguments. The primary functionalities of our containers are included in the file app.py. All our services are built on the Flask framework, and we will be gradually adding more layers of additional libraries and tools to meet the specific needs of each service as they become necessary.

| The backend endpoints of our web application **for** the service **A** (Alice) and **B** (Bob). We will be using only Alice source code since Bob's source code is identical.      from flask **import** Flask, send\_file, request **import** os from flask.wrappers **import** Response **import** pandas as pd **import** packages.etl\_pipeline as etl   HOST = '0.0.0.0' PORT = '9200' NAME\_OF\_CLUSTER = "Cluster\_A"   app = Flask(\_\_name\_\_)   @app.route("/take\_data/<matching\_field>/", methods=['GET']) @app.route("/take\_data/<matching\_field>/<int:noise>", methods=["GET"]) def **get**(matching\_field=None , noise=0):  """  The get endpoint/function that starts the ETL pipeline and return to   Carol the encrypted data    Returns:  http\_response: The file with the encrypted data  """    **if** 1000 <= noise <= 0:  **return** None    etl\_object = etl.ETLModel(matching\_field, noise)  etl\_object.start\_etl()  **return** send\_file(f'/var/lib/data/hidden\_information.csv',  mimetype='text/csv',  attachment\_filename='a\_cluster\_data.csv',  as\_attachment=True)   @app.route("/take\_data/", methods=["POST"]) def **post**():  """ The post endpoint/function that takes the data and stores them in the container    """ **try**:  downloaded\_data = pd.read\_csv(request.files['file'])  downloaded\_data.to\_csv('/var/lib/data/datafile.csv',  encoding='utf-8', index=False)   except FileExistsError:  **return** Response(f"We could not find the file!")  **else**:  **return** Response(f"Save the file")     **if** \_\_name\_\_ == '\_\_main\_\_':  ENVIRONMENT\_DEBUG = os.environ.get("DEBUG", True)  app.run(host=HOST, port=PORT, debug=ENVIRONMENT\_DEBUG) |
| --- |

*Code 13 The main code of Alice and Bob* 13

First and foremost, we must specify the IP address of the host as well as the port number on which our services will be accessible. In addition, we must decide on the names of our devices to have a clear idea of which service is responding. Finally, we must start our Flask object with the appropriate arguments to establish the development environment in which to conduct our experiment. Because this is the fundamental process for all of our microservices, we will not have to repeat the same information for each of them.

Our flask environment is now up and running, which means we are ready to process our data. Alice and Bob have source code that is quite similar since they do essentially the same purpose, thus we will show them in the same section.

Both Alice and Bob’s container is using two different endpoints. The first one, the post() method, will take the data from a source (e.g. API, dump files) and will save them in the storage. The second endpoint, get(), is the one that will start the ETL to apply noise and encrypt the data.

Carol can reach Alice and Bob with HTTP requests and since all three services are connected to the same docker network they can communicate with each other with the container name (e.g. cluster-a, cluster-b, jupyter). Therefore, when Carol requests the services of Bob and Alice it should reach the URL <http://cluster-a:9200/> and <http://cluster-b:9300/>. (cluster-a and cluster-b is the IP address of each container)

The experiment took place in two different environments. The development environment was the local machine (my personal computer) and the production environment. The first one was used to run locally at the location <http://localhost/>, all the upcoming changes of the code helping for a fast CI/CD. The second one was the live version of the app running on a Virtual Environment provided by the University of Macedonia at the URL 15.

| Function | Description |
| --- | --- |
| Get() | Start the ETL pipeline. Adds noise to the dataset and encrypts using Soundex and SHA. Finally, pushes the CSV file. |
| Post() | Acquired a CSV file and saves it in the storage |

Now that we have clarified the Alice and Bob service capabilities, we also have already sent encrypted data to Carol, and it is to match them. Before that, we will also explain the infrastructure of Carol’s container as well

4.9 Main Code – Carol

| **import** requests **import** datetime **import** pandas as pd from pyspark.sql **import** SparkSession       def **get\_data\_from\_service**(host, port, data, match\_field, noise):    requests.**post**(f"http://{host}:{port}//take\_data/", files=data)  request = requests.get(f'http://{host}:{port}//take\_data/{match\_field}/{noise}')  url\_content = request.content  with **open**(f"/opt/workspace/{host}\_download.csv", 'wb') as csv\_file:  csv\_file.**write**(url\_content)     def **main**(match\_field, noise, csv\_a, csv\_b, expected\_size):  """ The main code of Carol. It send data to Alice and Bob and then  it gets those data encrypted.    Args:  match\_field : the column which will be used to match the data  noise: the size of the fake noise we will add  csv\_a: Alice's data  csv\_b: Bob's data  expected\_size: the expected matches    Returns:  pd.Dataframe: All the metrics and times    """    start = datetime.datetime.now()    myfiles = {'file': open(csv\_a,'rb')}   get\_data\_from\_service(host='cluster-a', port='9200', match\_field=match\_field, noise=noise)  myfiles = {'file': open(csv\_b,'rb')}   get\_data\_from\_service(host='cluster-b', port='9300', match\_field=match\_field, noise=noise)    download\_time = datetime.datetime.now() - start        df\_1 = spark.read.csv(path="/opt/workspace/cluster-a\_download.csv", sep=",",   header=True)  df\_2 = spark.read.csv(path="/opt/workspace/cluster-b\_download.csv", sep=",",  header=True)    df\_1.createOrReplaceTempView("a\_cluster\_data")  df\_2.createOrReplaceTempView("b\_cluster\_data")      start = datetime.datetime.now() |
| --- |

| #Starts matching the Alice and Bobs data  matched\_data = spark.sql("SELECT a.NCID as a\_f1, a.first\_name as a\_f2,  a.last\_name as a\_f3, a.midl\_name as a\_f4,\   a.street\_name as a\_f5, a.res\_city\_desc as a\_f6, \  b.NCID as b\_f1, b.first\_name as b\_f2, b.last\_name  as b\_f3, b.midl\_name as b\_f4, b.street\_name as   b\_f5, b.res\_city\_desc as b\_f6 \  FROM a\_cluster\_data as a, JOIN b\_cluster\_data as b ON  a.first\_name == b.first\_name AND  a.midl\_name == b.midl\_name AND  a.street\_name == b.street\_name AND  a.res\_city\_desc == b.res\_city\_desc")      #saves the joined data in a csv file in Carols storage  matched\_data.repartition(1).write.mode('overwrite')\  .csv(f"/opt/workspace/joined\_result", header=True)    joined\_time = datetime.datetime.now() - start    matched\_data.createOrReplaceTempView("matched\_data")    #Finds the **true** positives of our experiment  TP = spark.sql(" SELECT \*\  FROM matched\_data \  WHERE matched\_data.a\_f1 == matched\_data.b\_f1 AND\  matched\_data.a\_f2 == matched\_data.b\_f2 AND\  matched\_data.a\_f3 == matched\_data.b\_f3 AND\  matched\_data.a\_f4 == matched\_data.b\_f4 AND\   matched\_data.a\_f5 == matched\_data.b\_f5 AND\  matched\_data.a\_f6 == matched\_data.b\_f6")    total\_matches = matched\_data.count()  TP = TP.count()  FP = total\_matches - TP  precision = TP / ( TP + FP )  recall = TP / expected\_size    **return** pd.DataFrame([[noise, download\_time, joined\_time, precision, recall,  total\_matches, expected\_size, TP, FP ]],\  columns=['noise', 'download\_time', 'joined\_time',\  'precision', 'recall','total\_matches',\  'expected\_size', 'TP', 'FP'])   #global pyspark object. spark = SparkSession.\  builder.\  appName("pyspark-notebook").\  master("spark://spark-master:7077").\  config("spark.executor.memory", "2g").\  getOrCreate() |
| --- |
|  |

*Code 14*

| # The main function of the container **if** \_\_name\_\_ == "\_\_main\_\_":    match\_field = 'NCID'  data = [(50, 12500), (100, 25000),(200,50000),(400,100000)]  **for** book, expected\_matches in data:  csv\_a = f'{book}K\_A.csv'  csv\_b = f'{book}K\_B.csv'  result = pd.DataFrame(None, columns=['noise', 'download\_time', 'joined\_time','precision', 'recall', 'total\_matches', 'TP' , 'FP' ])    **for** noise in **range** (0,1000,100):  result = pd.concat([result, main(match\_field, noise, csv\_a, csv\_b, expected\_matches)], axis=0)  result.to\_csv(f'final\_{book}.csv', encoding='utf-8', header=True, index=False) |
| --- |

*Code 15*

Carol's container is built using a Jupyter Notebook image as a starting point. Therefore, the source code and instructions are distinct from those used by Alice and Bob in their respective games. Below depicts the code that was used to request the data from Alice and Bob, as well as the Spark commands that were used to start the joined operation on the data from Alice and Bob.

Jupyter will serve as our primary platform for doing all of the required operations on the data we have obtained from Alice and Bob. To perform all of these tasks, we will need to import the modules that were utilized in the source code, just like we did in the previous containers. Finally, we will present the Spark program, which will be used to complete our unionization process. We will need to download the data from Alice and Bob after we have created our Spark Session with the correct attributes and configuration settings, such as whether it will run in a normal stand-alone mode or a cluster mode.

Apache Spark will be used to do our data analysis on our datasets. We will also perform the join operation on our datasets using Apache Spark to see how accurate our implementation is based on our predictions. We will be using Spark since we are expecting it to return fast and accurate results.

| Functions | Description |
| --- | --- |
| get\_data\_from\_service() | It gets the data from Alice or Bob which are encrypted with Soundex and SHA. |
| main() | The main code where the join operation takes place |

We will now start explaining the processing steps that Carol needs to take to perform our operation but before all that, we need to start the pyspark cluster object which will perform all the operations. We need to connect the pyspark object with the distributed cluster and add configuration options to optimize the processing of the data.

The first step is for Carol to request Alice and Bob to send their data. These data have already been transformed, with Soundex and SHA, and therefore they will be received by Carol in that specific form so that protection and anonymity will be sustained. After we have received the data, we should move them in a structure by creating pyspark dataframes. We will be using two different pyspark dataframes, one for the data of Alice and one for the data of Bob. After Carol has received the data from both sides, she creates the dataframes and she will start to perform the join operation upon them.

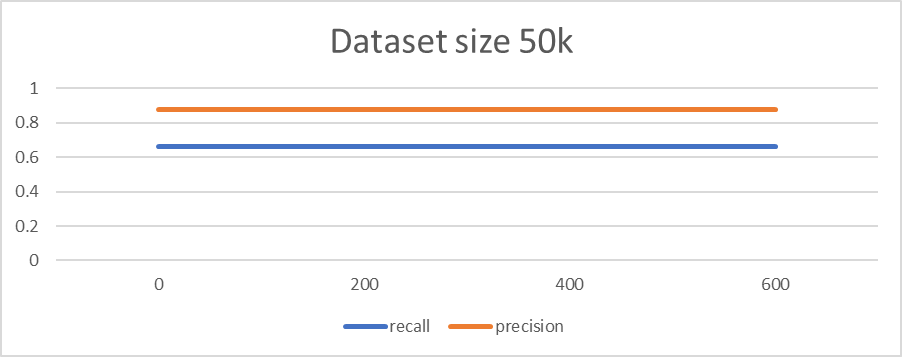
Both dataframes have identical fields (NCID, res\_city,\_desc, street\_name, midl\_name, first\_name, last\_name ) that will be used in the join operation. Except for the field with the name “NCID” which is being used later as a validator (fake data or real data) while all the other fields are being used to identify possible matches. As mentioned, we will be using the NCID to find the parameters of our metrics.

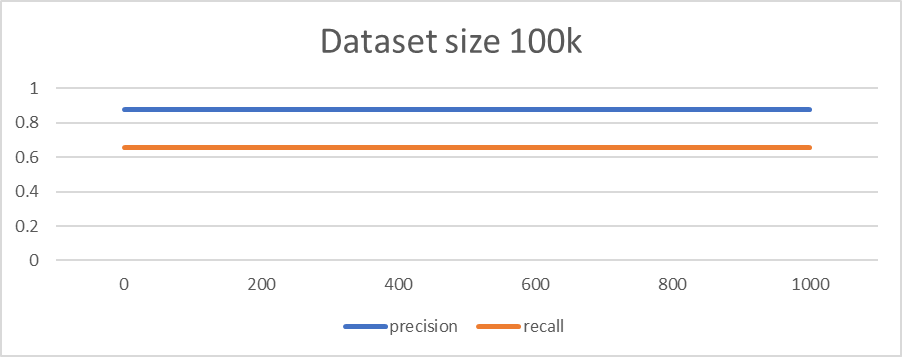
We will be using for our example two different statistical metrics precision and recall. **Precision** (also called positive predictive value) is the fraction of relevant instances among the retrieved instances, while **recall** (also known as sensitivity) is the fraction of relevant instances that were retrieved. Both precision and recall are therefore based on relevance [‎39].

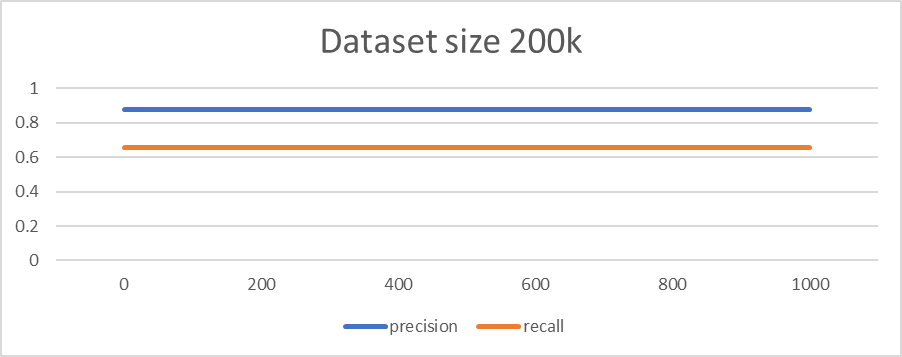
5. ***Empirical Evaluation***

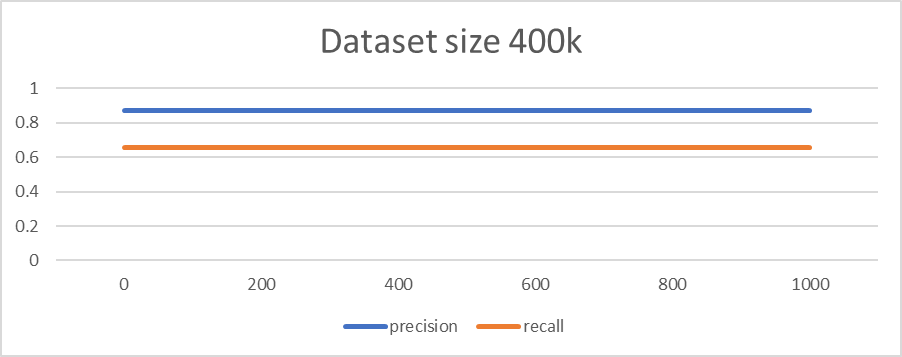
We will be using four different datasets regarding their size (50k, 100k, 200k, and 400k). We will present the calculated recall and precision and also some of the other metrics required to have a deep understanding of our experiment’s behavior and accuracy.

The first quartets of graphs are showing the accuracy and recall of our experiment regarding the size of the datasets and also the size of the noise added to the dataset. The second quartets of chars present the time spent while downloading/transforming the data and the time required for the join operation.

*Figure 1: Presents the recall and precision after the join operation on the 50k size datasets*

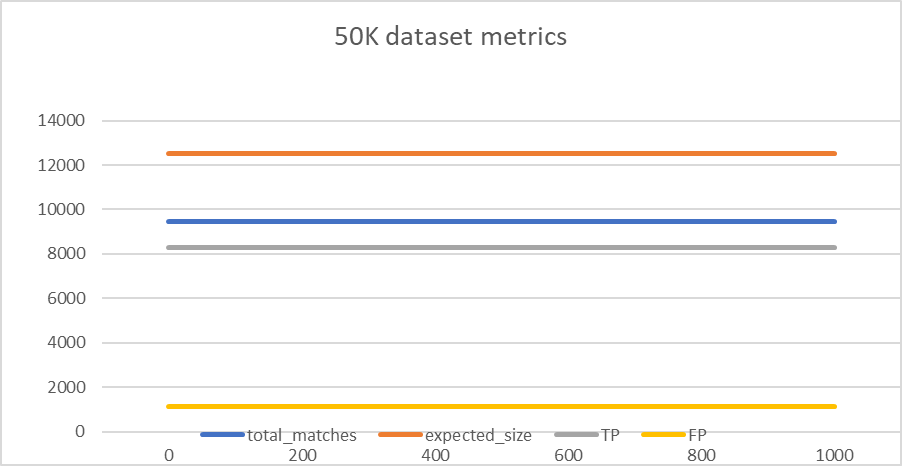
*Figure 2: Presents the recall and precision after the join operation on the 100k size datasets*

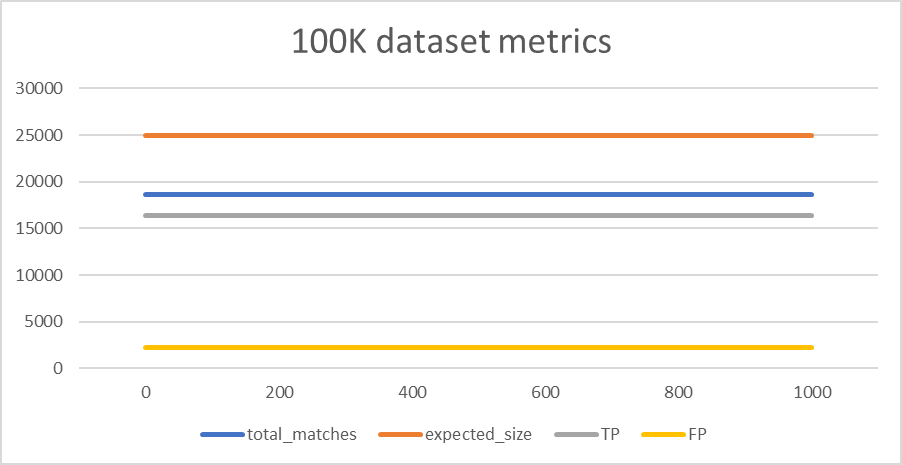
*Figure 3: Presents the recall and precision after the join operation on the 200k size datasets*

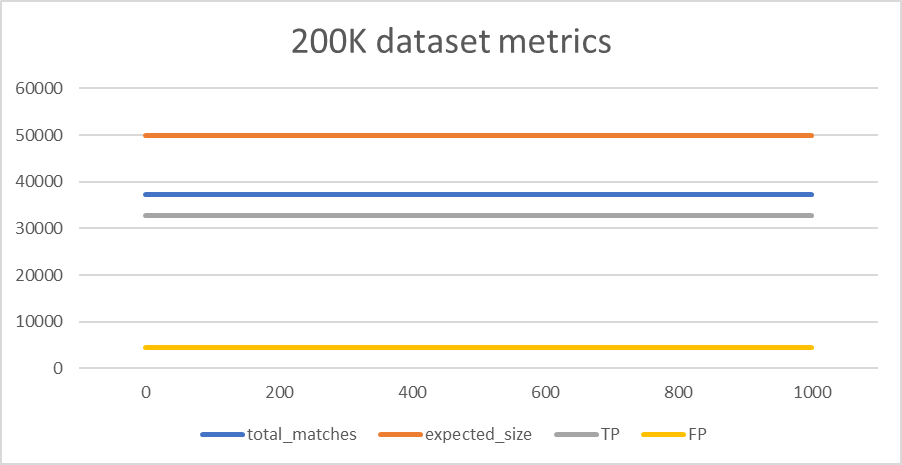
Figure 4: Presents the recall and precision after the join operation on the 400k size datasets

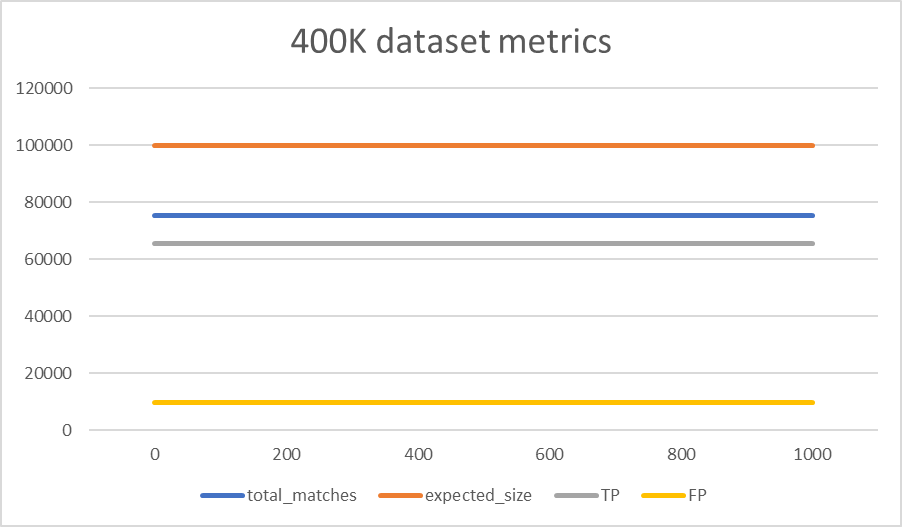


Now we will present all the metrics that are required for our statistical metrics.

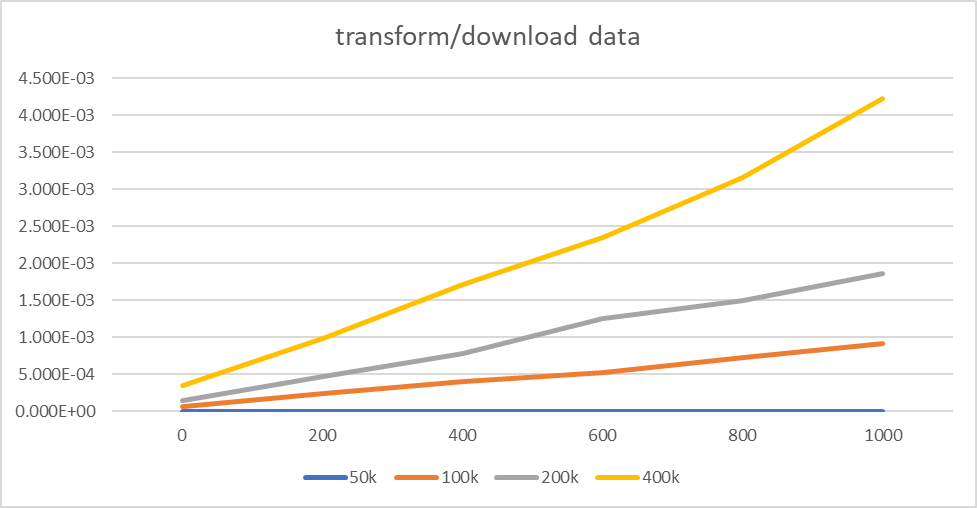
*Figure 5: Presents the metrics of the datasets 50k. total\_matches: the total matches of the join operation, expected\_size: is the expected size of the matches, TP: the true positive values, FP: the false*

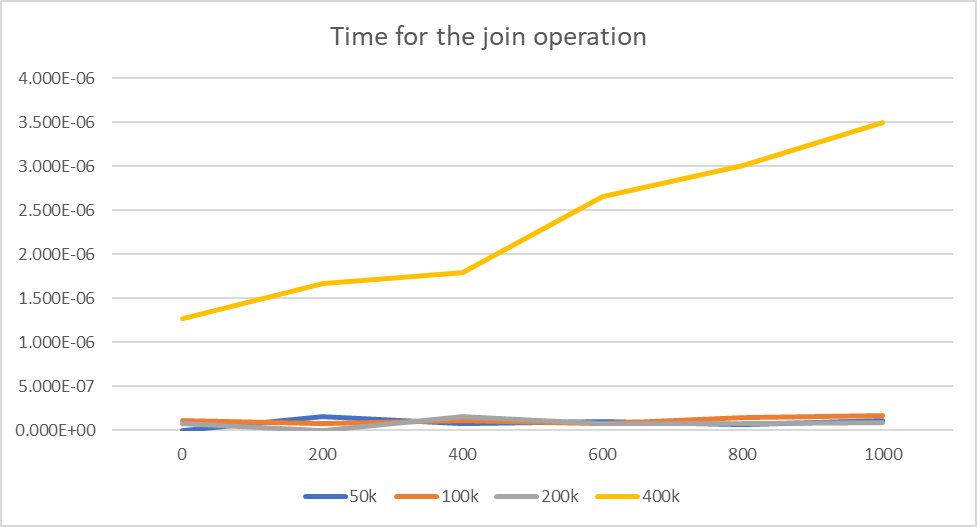
*Figure 6: Presents the metrics of the datasets 100k. total\_matches: the total matches of the join operation, expected\_size: is the expected size of the matches, TP: the true positive values, FP: the false positive values*

*Figure 7: Presents the metrics of the datasets 200k. total\_matches: the total matches of the join operation, expected\_size: is the expected size of the matches, TP: the true positive values, FP: the false positive values*

Figure 8: Presents the metrics of the datasets 400k. total\_matches: the total matches of the join operation, expected\_size: is the expected size of the matches, TP: the true positive values, FP: the false positive values

The last four graphs present the time needed in very key parts of our application. The first part is the transformation and download time from the Alice and Bob container and the second one is the time elapsed when trying to perform the join operation in our spark cluster.

*Figure 10: Presents the total time required for the services to transform and download the data from one service to another on the different sizes of datasets.*

*Figure 11: Presents the time needed to perform the join operation with the pyspark on the different sizes of datasets*

We have now concluded our experiment section, which demonstrated how it is feasible to do the PPRL using Soundex while utilizing Microservices Oriented Architecture. As previously said, this is just the first milestone in our project's development, as it only consists of a very simple design. Even though the project's complexity does not approach the complexity of the real-world issues, we can see how advantageous it is to work in this architecture. Adding additional microservices and adopting a new spark method will be discussed so that we can have a better understanding of what more could be done to optimize our application, which will serve as the conclusion to this thesis.

***6***. ***Conclusion and future work***



As we have previously stated, new technologies arise throughout time, and we must develop a strategy for our application that allows for simple adaptation and development. We have demonstrated the advantages and disadvantages of using phonetic algorithms with the microservices oriented architecture. It has also been demonstrated that it is a strategy that should be implemented for all future "big" applications because it is beneficial to their survival. The significance of Privacy-Preserving Record Linkage to keep all of the transferring information safe and concealed inside of the microservices communication channel was also shown. We've also proved how easy and safe it is to integrate applications with new frameworks using microservices, as well as to remove or update existing ones. Most significantly, we have shown how applications may be impacted by the implementation of new frameworks by carefully monitoring the behavior of our project after the integration time has ended. As a bonus, we've shown the advantages for novice developers who may be involved in the development of applications at any moment since it creates a healthy and inviting atmosphere that prevents them from being overwhelmed by the complexity of monolithic programs. Finally, and perhaps most elaborately, we have discussed the financial advantages that businesses get from this design since it is often the first variable that organizations consider before implementing any significant changes. The use of this architecture helps industries and companies that are striving for success and maintaining a solid position in a competitive market, thus avoiding losses that would otherwise jeopardize the integrity of an enterprise.

As a novice programmer and developer, I had to deal with the majority of the problems that monolithic-based applications brough. There were no significant issues with the time commitment I made to understand the theoretical model of MOA as well as the execution portion of the experiment. All of the frameworks that were utilized offered very detailed documentation papers that assisted inexperienced developers, like me, in breaking into the field, particularly the data firm on which this thesis concentrates. After everything is said and done, more time has been spent learning all the frameworks used in the application rather than learning the application of MOA, which serves as a compelling justification for continuing to adopt this approach in the future.

The next step of this paper is to optimize all the methods and tools used to provide results faster.

***7. BIBLIOGRAPHY AND REFERENCES***

1. *Rodgers, Peter. “Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity Web Services Edge 2005 East: CS-3. CloudComputingExpo 2005. SYS-CON TV. Archived from the original on 20 May 2018. Retrieved September 2021*
2. *Zaharia, Matei;Chowdhury, Mosharaf; Franklin, Michael J.; Shenker, Scott; Stoica, Ion.* [*Spark: Cluster Computing with Working Sets*](https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf) *(PDF). USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*
3. *Odell, M., Russell, R.C.: The Soundex coding system. US Patents, 1261167 (1918)*
4. *Alexandros Karakasidis,* [*Georgia Koloniari*](https://dblp.org/pid/90/4426.html)*:Phonetics-Based Parallel Privacy Preserving Record Linkage.* [*3PGCIC 2017*](https://dblp.org/db/conf/3pgcic/3pgcic2017.html#0001K17)*: 179-190*
5. *Alexandros Karakasidis, Vassilios S. Verykios:Privacy Preserving Record Linkage Using Phonetic Codes.* [*BCI 2009*](https://dblp.org/db/conf/bci/bci2009.html#KarakasidisV09)*: 101-106*
6. [*Peter Christen*](https://dblp.org/pid/c/PeterChristen.html)*: A Comparison of Personal Name Matching: Techniques and Practical Issues.* [*ICDM Workshops 2006*](https://dblp.org/db/conf/icdm/icdmw2006.html#Christen06)*: 290-294*
7. *Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges, 2017, Dinusha Vatsalan, Ziad Sehili, Peter Christen, Erhard Rahm* [*(anu.edu.au)*](https://openresearch-repository.anu.edu.au/handle/1885/247396)
8. *Implementing a hash-based privacy-preserving record linkage tool in the OneFlorida clinical research network by Jiang Bian, Alexander Loiacono, Andrei Sura, Tonatiuh Mendoza Viramontes, Gloria Lipori, Yi Guo, Elizabeth Shenkman, and William Hogan*
9. *Securing Bloom Filters for Privacy-Preserving Record Linkage by Thilina Ranbaduge, Rainer Schnell.*
10. [*Privacy-Preserving Record Linkage in the context of a National Statistics Institute - GOV.UK (*](https://www.gov.uk/government/publications/joined-up-data-in-government-the-future-of-data-linking-methods/privacy-preserving-record-linkage-in-the-context-of-a-national-statistics-institute)[*www.gov.uk*](http://www.gov.uk)[*)*](https://www.gov.uk/government/publications/joined-up-data-in-government-the-future-of-data-linking-methods/privacy-preserving-record-linkage-in-the-context-of-a-national-statistics-institute)
11. [*Peter Christen*](https://dblp.org/pid/c/PeterChristen.html)*,* [*Rainer Schnell*](https://dblp.org/pid/127/7152.html)*,* [*Dinusha Vatsalan*](https://dblp.org/pid/89/7958.html)*,* [*Thilina Ranbaduge*](https://dblp.org/pid/162/3257.html)*: Efficient Cryptanalysis of Bloom Filters for Privacy-Preserving Record Linkage.* [*PAKDD (1) 2017*](https://dblp.org/db/conf/pakdd/pakdd2017-1.html#ChristenSVR17)*: 628-640*

END OF THESIS