APPLIED INFORMATICS

BIG DATA

Privacy-preserving  
record linkage using microservices

Assessment

Konstantinos Razgkelis



ABSTRACT

The exponential growth of the technological sector, in addition to an increasing need for sophisticated structures, necessitates a revamping of the current methods. Many organizations are putting a microservices-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 article lays out the advantages of the microservice architectural design, while developing a system that uses many different frameworks to prove the benefits of MOA. 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 in order to build new systems with the application of modern technology.

ACKNOWLEGEMENTS

Before I begin the explanation of this paper, I would like to express my gratitude to Mr. 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 want 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 to me.

CONTENTS

1. **Microservices Oriented Architecture**  
   1.1 Introduction to Moa

1.2 What is a microservice

1.3 MOA: Pros

1.4 MOA: Cos  
1.5 MOA: Conclusion

2. **Docker**  
 2.1 Enter Docker  
 2.2 Docker: Introduction   
 2.3 Docker: Images  
 2.4 Docker: Containers  
 2.5 Docker: Volumes  
 2.6 Docker: Compose  
 2.7 Docker: Docker Hub and repositories  
 2.8 Docker: The last word

3. **Apache Spark**  
 3.1 Apache Spark: Introduction  
 3.2 Apache Spark: Foundation  
 3.3 Apache Spark: Resilient Distributed Datasets (RDDs)  
 3.4 Apache Spark: Apache Spark over Apache Hadoop  
 3.5 Apache Spark: lazy evaluation  
 3.6 Apache Spark: Apache Spark and Clustering

4. **Jupyter Notebook**  
 4.1 Jupyter Notebook: Introduction  
 4.2 Jupyter Notebook: Data Science and Jupyter  
 4.3 Jupyter Notebook: Benefits  
 4.4 Jupyter Notebook: Conclusion

5. **Phonetic Algorithm**  
 5.1 Phonetic Algorithm: Introduction  
 5.2 Phonetic Algorithm: Soundex  
 5.3 Phonetic Algorithm: Example  
 5.4 Phonetic Algorithm: Privacy Preserving Record Linkage Using Phonetic Codes

6. **Python**   
 6.1 Python: Prologue  
 6.2 Python: Why Python  
 6.3 Python: Modules  
 6.4 Python: Flask  
 6.5 Python: Pandas  
 6.6 Python: Requests  
 6.6 Python: Jellyfish   
 6.7 Python: Conclusion

7. **Experiment**  
 7.1 Experiment: Prologue  
 7.2 Experiment: Introduction  
 7.3 Experiment: docker-compose.yml  
 7.4 Experiment: Dockerfile-Alice and Bob  
 7.5 Experiment: Dockerfile-Carol  
 7.6 Experiment: Main Code – Alice and Bob  
 7.7 Experiment: Main Code – Carol  
 7.8 Experiment: Conclusion

8. **Conclusion** 8.1 Conclusion: Epilogue  
 8.2 Conclusion: My Final Thoughts

9**. Future Enhancements and Improvements**

9.1 Updates

10**. Bibliography and References**

1. INTRODUCTION

Many diverse resources have been utilized from the beginning of human civilization for the purposes of financial stability, economic interaction, and a fair market, all with the goal of establishing a healthy system that would allow for long-term development. Additionally, prices varied based on the availability and demand for those resources, as well as the significance of those resources in meeting human needs. For the majority of the nineteenth century, solid commodities such as gold and silver were considered the most valuable resources; but, with the discovery of oil, the economy shifted its focus to a new direction. In conclusion, in order to continue to live and flourish in contemporary times, all of the colossal industries must have adopted tools, and the majority of these tools are concerned with data. As a result, it is reasonable to conclude that data is today's most valuable resource, and the person who owns the data controls the whole universe.

The unfortunate reality is that dealing with data is not a simple process. By themselves, the sheer volume of data that all companies produce at the same time is a difficult job, and it is also conceivable to transmit an identity gateway while sharing this information with other sectors. In certain cases, data such as names, addresses, and ID numbers are filtered so that they cannot be viewed, but other less "essential" variables may be used in specific methodologies to allow the identification of the 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 implementation of GDPR regulations, more stringent restrictions have been implemented in order to enhance the privacy of data that is sent to third parties who should not be able to identify people with the data being sent. The Record Linkage Problem is a term that is frequently used in the literature to express what we have just explained. 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 while without disclosing any more information to one other, they are referred to as the Privacy Preserving Record Linkage issue. With the introduction of errors, especially from human’s interaction, can make this problem exponentially harder.

Additionally, all the organizations previously mentioned have developed apps that are equipped with tools that are capable of carrying out the necessary activities in order to fully utilize the data that has been produced. Naturally, in order to remain competitive in the market, every application must be updated on a regular basis in response to market needs. Whether a big or small update is coming, it is essential that you have a thorough knowledge of the application's functioning, or else it may cause issues with the system's overall integrity. In certain cases, this may be dangerous depending on the size of the program, since larger applications find it more difficult to adopt newer technologies because it involves cleaning up or changing existing code. If the transplanting techniques used are accurate and appropriate, the program may be updated without any problems. 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.

In this post, we will 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 join in 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 with using it. As an added bonus, we will combine these methods into one experiment in order to demonstrate the advantages of these tactics when correctly executed together in terms of system performance and accuracy of the findings. In order to put these theoretical models into reality, we will also make use of appropriate contemporary technologies, such as Apache Spark and Docker, among other things. Finally, we offer comprehensive examples in order 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

As clearly mentioned in the [13],

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

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

* 1. Monolithic Architecture

In software development, 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 is consisted by a uniform approach were 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 amazing number of code lines that are all maintained in a single system. As a result, in order to introduce new frameworks or to resolve some of the debugging problems that may arise, an enormous amount of time and human resources would be required.

Despite this, there are certain unique reasons that may make this method particularly helpful since it may assist specific applications, often 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 does 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 he 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 transplant occurring in every current application is quite low, owing to the fact that 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 of time in order to adapt to the new age is just not financially feasible. The technique for this "surgery" is extremely time-consuming, and as a result, there will always be some "scars" 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.

Getaway from the monolithic era

During the year 2005 [1], the phrase "Microservices Oriented Architecture" 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, 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.

3.2 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 own 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 own authority and views the rest of the external services as black boxes that can only be accessed via APIs. Let's be clear: although if MOA portrays the microservices as totally distinct and independent schemes, they are not completely isolated from one another in terms of functionality. A link between them is established via the use of dump pipes or lightweight protocols such as REST API [2].

  Let us now see the benefits offered by adopting the microservices-oriented architecture, as outlined in [33].

MOA: Pros

**Individualism**: 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 order for the duties of each service routine not to interfere with the activities of the others, they are constructed in such a manner that they do not conflict with one another. 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. This also demonstrates the adaptability of this approach when it comes to the goal of back-up and recovery protection.

As previously said, this architecture is not restricted to any one geographical area. Each microservice may be developed and delivered in a variety of geographical locations, including, of course, any location in the globe. For each microservice, this phenomenon articulates the concept of individuality even more clearly, and it begs the issue of how essential it is for them to communicate with one another.

**Scalability**: As the success of an application grows, so do the demands on the application's size. Due to the fact that each microservice functions as a separate department, each microservice has a unique set of resource requirements. Because 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. Microservices Architecture 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. The resources are allocated precisely according to the requirements, and every growth is a financially feasible and risk-free investment.

**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 due to the fact that 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 maintainers team to grasp the code, enabling new and inexperienced developers to join in the project, resulting in quicker development and a simpler and more inexpensive to maintain codebase overall.

**Failure Isolation**: A flawless project, with no minor or significant mistake, could only exist in a utopian or dystopian society. The presence of a bug may have a negative impact on 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.

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 adopt in order 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 completely eliminated. However, this is a philosophical quandary that belongs in 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.

**Communication Difficulty**: 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 with one another 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.

**Implementation and resource distribution**: Each microservice has its own database, which needs a distinct set of resources to maintain and carry out its functions properly. It is necessary to have experienced developers present in order 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 by. 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.

**Correction of Errors and Debugging**: 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 own 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.

**Integration**: 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.

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 in a safe manner while maintaining the benefits of both methods being emphasized.

MOA and Monolithic with an example

When describing a monolithic design, you may define it as a black box that is capable to perform the functions of a car. However, the description of a microservice architecture might be defined as a group of black boxes that are capable to accomplish 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 appropriate collaboration.



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

3.3 Enter Docker

The first chapter has shown some of the most often used reasons in support of the use of a MOA-based application design. On the basis of these assumptions, we will investigate Docker, a piece of software that has many usage capabilities, the most significant of which for the purposes of this article is the ability to put the MOA into action in a real-world environment. [3]. Docker is a collection of platforms as a service (PaaS) technology that distribute software in packages known as containers by using operating system-level virtualization [4]. Docker is made up of many components such as images, containers, services, networks, volumes, and plugins. [5]

Docker: An Overview

Utilizing Docker, which is an easy-to-learn and fast-deploy platform, it is simple to create and operate an application using the MOA framework. Docker comes with a very comprehensive documentation package that contains more than enough information to assist developers of varying levels of expertise in participating in this integration. Docker's ability to handle a wide range of programming languages, including C/C++, Clojure, Go, Hy, Java, Node, Perl, PHP, Python, Rails, and Ruby, is a significant advantage. Container applications are created and delivered via the Docker Hub, which is an online service provided by Docker that allows a group of developers to utilize, build, manage, and distribute container applications. Furthermore, images may be found and shared utilizing the world's biggest container repository, which is available for free [6].

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 in order 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 [21].

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 own 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. 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 [22].

Containers have the wonderful 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 [22].

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. Despite the fact that 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 [7].

Docker: Compose

As previously stated, the Docker API and Docker CLI commands may be used to construct any and all of Docker's objects. Docker includes another useful tool for developers, enabling them to create more quickly and easily than ever before. A tool for creating and executing multi-container Docker applications, Docker-compose is available for download here (8). Compose makes use of a particular YALM file that configures all of the information required to build and run the application in question. Information about container characteristics, such as the container's unique name inside Docker, the IP address, the relationship with linked volumes and networks, and the container's size, are provided via the container's YALM file [8].

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 with the goal of ensuring steady and fast expansion. 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 [8].

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 [6].

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.

3.4 Privacy Preserving Record Linkage (PPRL)

The following mathematical formula, theory and example is used to explain the problem we are trying to cope and it is taken from [Karakasidis A. ,Koloniari G. : Phonetics-Based Parallel Privacy Preserving Record Linkage , University of Macedonia, Thessaloniki, Greece].

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 [12].

Alice and Bob will most probably use different schemas in their databases. As such, they will have different attributes. Let RA be Alice’s schema and RB be Bob’s schema and let us assume that in these schemas m of the attributes are common between the two sources forming a composite key. These attributes might be names, surnames, addresses, 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.

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.

5.1 Phonetic Algorithm: Introduction

A phonetic algorithm helps standardize the way people speak in a specific dialect. It's been well over a century since the original algorithm Soundex was created [12], 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 [13].

5.2 Phonetic Algorithm: Soundex

Soundex [24] 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.

5.3 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."

|  |  |
| --- | --- |
| 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

5.4 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 [14]. 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.

In order to comprehend how PPRL with Soundex works, we will be using the same example used in the paper [13] 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 join 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 [14].

1. Their data is changed using a Map function in order 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.

Chapter 3

Apache Spark

3.1 Apache Spark: Introduction

Apace 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 [10].

3.2 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 [9].

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 [9].

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 [9]. It is possible to use the same code for batch processing and real-time streaming applications, which improves the productivity of the developer.

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.

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 [9].

Diagram, shape

Description automatically generated

Figure 2 A graphical presentation of the Apache Spark software.  
source: [ 9 ]

3.3 Spark: Resilient Distributed Datasets (RDDs)

In computing, a resilient distributed dataset (RDD) is a read-only collection of objects that has been partitioned over a number of computers and that can be recreated if a partition has been lost [10]. 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.

3.4 Apache Spark: Apache Spark over Apache Hadoop

Using a parallel, distributed approach, Hadoop MapReduce is a programming tool for processing 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 [9].

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. Due to the fact that 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.

3.5 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 [11].

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. Despite the fact that 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 [23].

Chapter 7

Introduction to the problem

7.1 Experiment: Prologue

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 microservices-oriented architecture, and a number of 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.

7.2 Experiment: Introduction

First and foremost, we must address our microservices and the responsibilities they bear. In this experiment, we will create a fundamental 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.

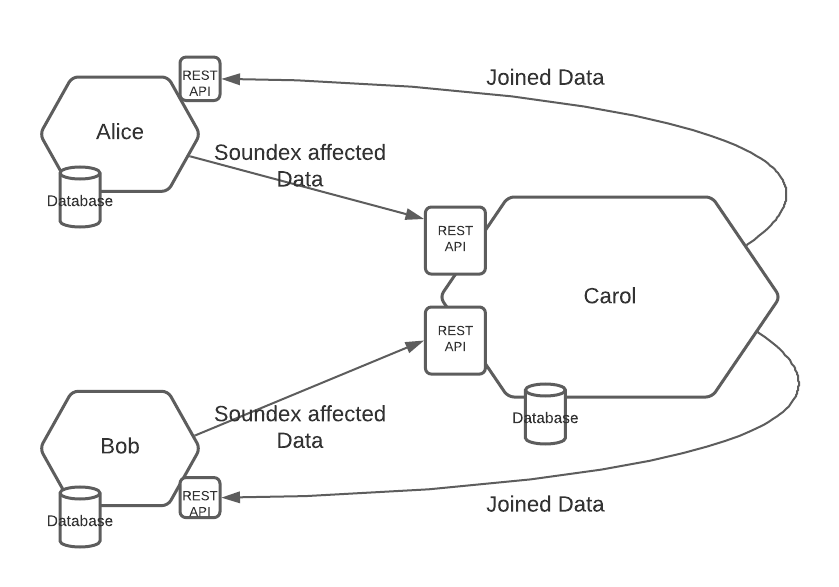


Figure 3

7.3 Experiment: docker-compose.yml

As we previously said in chapter 2, Docker is the primary program with which we will integrate all of 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 in order to proceed. This file contains an explanation of all of the configuration options that have been applied to the microservices. In figure 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 [31]:

In the section “version” we define the version of the .yml file since different version requires different commands for the proper configuration

version: '2.0'

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: Alice

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: Bob

build:

context: ./services/data-service-2

dockerfile: Dockerfile

ports:

- "9300:9300"

environment:

- PORT=9300

volumes:

- cluster-B-volume:/var/lib/data

jupyterlab:

container\_name: Carol

build:

context: ./services/jupyter

dockerfile: Dockerfile

ports:

- 8888:8888

- "4040-4080:4040-4080"

volumes:

- shared-workspace:/opt/workspace

Code 1 . This is the docker-compose.yml files.   
source: [30]

In the section “volumes " we define the volumes that we will be 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:

‘‘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 2, since we don’t need to implement the Version 3 functions.

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

In order to properly develop the experiment, it is essential to refer to each container by its own unique and particular identification number. 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.

The command "build" does exactly what it says it will do: 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 on the basis of 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 on the basis of 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 in order 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. Volumes, as shown in the code's last lines, may be generated quickly and easily by just declaring their names. A number of 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 in order 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.

7.4 Experiment: Dockerfile-Alice and Bob

In our last discussion, we said that every container could be built on the basis of 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. In order to do this, we have created a Dockerfile, which we will describe in detail the commands that are used to configure it.

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 in 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 Figure 5 we can see Alice and Bob Dockerfile [32][33].

Defines the image of which the container is going to be based

FROM python:3

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 --no-cache-dir -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 2. This is the Docker file of Alice and Bob.  
source: [31][32]

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 in order for the container to do its own 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.

7.5 Experiment: Dockerfile-Carol

This is the image the container is based of

FROM ubuntu:latest

Here it defines the user of the Linux OS

USER root

The libraries and dependencies required

COPY requirements.txt ./

We run all the OS commands here

RUN apt-get update && apt-get -y update

RUN apt-get install -y build-essential python3.6 python3-pip python3-dev

RUN pip3 -q install pip --upgrade

RUN pip3 install --no-cache-dir -r requirements.txt

RUN pip3 install wget

RUN pip3 install pyspark

RUN pip3 install jupyter

USER $NB\_UID

We start the main application using cmd

CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0", "--allow-root"]

Code 3. This is the Dockerfile for Carol.  
source: [32]

It is necessary for us to utilize an operating system on which our container will be built, just as Alice and Bob did. Instead of displaying the Python image in this container, we will attempt to demonstrate a more manual method. We will be utilizing the basic and most recent version of the Ubuntu image as our operating system, and we will be using the command "FROM" once again. In order for our container to function properly, we will also need to manually implement Python and all of its libraries, since our container would otherwise only be capable of running the most basic Linux commands.

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 requiriemnt.txt file once again using the “COPY” command, and we will need to execute certain OS commands to complete the installation of the prerequisites, just as we did with the previous containers. In this part, we'll go through how to manually download Python using Sudo commands, as well as how to download certain specific modules. We will eventually download Jupyter notebook and Spark, and we will be able to tell the container which application to start by using the “CMD” command at the end of this process. As a result, it will begin with the commands "Jupyter" and the word "notebook" in port number 8888 at the localhost IP address.

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.

7.6 Experiment: 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. Figure 7 depicts the source code, and we will begin to describe its operation after that point.

Code 4

HOST **=** '0.0.0.0'

PORT **=** '9200/9300'

NAME\_OF\_CLUSTER **=** "Alice/Bob"

app **=** Flask(\_\_name\_\_)

**if** \_\_name\_\_ **==** '\_\_main\_\_':

ENVIRONMENT\_DEBUG **=** os**.**environ**.**get("DEBUG", **True**)

app**.**run(host**=**HOST, port**=**PORT, debug**=**ENVIRONMENT\_DEBUG)

All of 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 available. 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 via a gateway. In addition, we must decide on the names of our devices in order to have a clear idea of how they will be used throughout their implementation. Last but not least, we must start our Flask object with the appropriate parameters in order 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 service environment is now up and running, which means we will need to add additional functionality to it in the future. Alice and Bob have code that is quite similar since they do essentially the same purpose, and we will show them in the same section. The following are the fundamental roles of Alice and Bob, and could be easily seen in the following Figure 8:

@app**.**route('/')

**def** get():

**return** f'{NAME\_OF\_CLUSTER}', 200

@app**.**route("/take\_data/<noise>", methods**=**["GET"])

**def** post(noise):

**try**:

noise **=** int(noise)

**except** ValueError:

**return** Response('<html><head><h1 style="background- color:powderblue;">There was an error!</h1></head></html>', 400)

**else**:

print(f"{noise}")

**if** 0 **<=** noise **<=** 100:

print(f"{NAME\_OF\_CLUSTER}- Data is being send")

**try**:

df1 **=** pd**.**read\_csv("/var/lib/data/A\_1k\_names\_separated.csv", header**=**0, names**=**[0,1,2])

column\_1 **=** df1[0]**.**apply(**lambda** x: jellyfish**.**soundex(x))

column\_2 **=** df1[1]**.**apply(**lambda** x: jellyfish**.**soundex(x))

column\_3 **=** df1[2]**.**apply(**lambda** x: jellyfish**.**soundex(x))

**except** Exception **as** e:

**return** Response('<html><head><h1 style="background-color:powderblue;">There was an error!</h1></head></html>', 400)

**else**:

data\_list **=** list()

**for** \_ **in** range(int(noise **\*** df1**.**shape[0] **/** 100)):

data\_list**.**append( [jellyfish**.**soundex( create\_alp()) **for** \_ **in** range(df1**.**shape[1]) ] )

df2 **=** pd**.**DataFrame(data\_list, columns**=**[0,1,2])

df2**.**to\_csv("/var/lib/data/noise.csv", encoding**=**'utf-8', index**=False**)

result **=** pd**.**concat([column\_1, column\_2, column\_3], axis**=**1)

result **=** pd**.**concat([result,df2], ignore\_index**=True**, axis**=**0)**.**sort\_values(by**=**2)

result**.**to\_csv('/var/lib/data/joined\_data.csv', encoding**=**'utf-8', index**=False**)

print(f"{NAME\_OF\_CLUSTER}- the download has finished")

**else**:

data\_list **=** list()

**for** \_ **in** range(int(noise **\*** df1**.**shape[0] **/** 100)):

data\_list**.**append( [jellyfish**.**soundex( create\_alp()) **for** \_ **in** range(df1**.**shape[1]) ] )

df2 **=** pd**.**DataFrame(data\_list, columns**=**[0,1,2])

df2**.**to\_csv("/var/lib/data/noise.csv", encoding**=**'utf-8', index**=False**)

result **=** pd**.**concat([column\_1, column\_2, column\_3], axis**=**1)

result **=** pd**.**concat([result,df2], ignore\_index**=True**, axis**=**0)**.**sort\_values(by**=**2)

result**.**to\_csv('/var/lib/data/joined\_data.csv', encoding**=**'utf-8', index**=False**)

print(f"{NAME\_OF\_CLUSTER}- the download has

Code 5

Using clear URLs in modern online apps helps users understand what's on each page. If people have URLs they can remember and utilize to go to certain web pages, they are more likely to like those pages and revisit them more often. Therefore, for the purpose of this experiment, we had to create two different methods, one that simply returns the name of the current microservice and the second the encrypted data affected by Soundex, noise, and SHA256 algorithm.

A positive response code is returned by the first method, "name()," which returns the name of the cluster. There is nothing more to say about this method since its functionality is very basic and straightforward.

“data()” is the second method in the main code of our services that describes the operation of Alice and Bob. In order to get access to this function of the service we need to use a specific URL path like <http://localhost:port/take_data/noise> and by using this we can creally see some of the necessary fields that are created (LOCALHOST, PORT, TAKE\_DATA, NOISE). We will explain in detail what are these fields and how they work.

“LOCALHOST” is defined at the start of the code which denotes the domain address of the application. In our project, the IP address which is declared is “0.0.0.0” that corresponds to the host’s IP address “snf-21129.ok-kno.grnetcloud.net” provided by [34].

Now each service is listening to different port numbers, Alice at “9200” while Bob at “9300”. Therefore, in the “PORT” field we need to enter the correct port number depending on which service we want to use. After we have announced which service will be used we need to also declare which function.

The “TAKE\_DATA” field is simply a GET request which returns a CSV file that contains the data after they have been effected by the Soundex method. The last field “NOISE” is an argument for the “TAKE\_DATA” request call which declares the percentage of fake Soundex objects will be shuffled in our main data list. Specifically, noise is an integer value that indicates the proportion of fabricated Soundex findings that will be included in our data set.

The Soundex algorithm will be applied to the data from the path now contained inside the container (path="/var/lib/data/data\_with\_names.csv"). If all of the parameters are valid, this service will begin applying the Soundex algorithm to the data from the destination. Following the completion of this procedure, the service will begin creating fake Soundex objects based on the specified percentage (for example, if we have a data list with 1000 rows of different data, and the noise is 20%, the service will generate 200 fake Soundex results), and at the end of the procedure, it will combine these two DataFrames into a single concrete object. Later, it makes use of a function from the hishlab library, which hashes the information using the SHA256 method, and lastly, it arranges the results of the hashing in a particular sequence in order to mix the actual Soundex data with the fictitious data.

If the above-mentioned procedure is performed successfully and without error, the dataframe is eventually written to a CSV file, which is then transferred to a different server inside our MOA program for further processing.

Additionally, there are basic HTML replies available in each instance, which give the user a message stating the service's action. This is not really a mandatory procedure, because our services will not require that information in order to perform their duties, and because it would result in a larger header in our communication system, but it was necessary for the development of the system because it provided some basic information about the services' behavior.

Now that we have clarified the Alice and Bob service capabilities, we have already sent encrypted data to Carol in order for her to match them.

7.7 Experiment: Main Code – Carol

Carol's container is built using the 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. Code 6 6 depicts the code that was used to request the data from Alice and Bob, as well as the Spark commands that were used to combine the data from Alice and Bob.

**import** pyspark

**import** requests

**import** pandas **as** pd

**import** hashlib

**from** pyspark.sql **import** SparkSession

spark **=** SparkSession**.**builder \

**.**master("local") \

**.**appName("PPRL") \

**.**getOrCreate()

request **=** requests**.**get(f'http://snf-21129.ok-kno.grnetcloud.net:9200//take\_data/')

url\_content **=** request**.**content

csv\_file **=** open("a\_download.csv", 'wb')

csv\_file**.**write(url\_content)

csv\_file**.**close()

request **=** requests**.**get(f'http://snf-21129.ok-kno.grnetcloud.net:9300//take\_data/')

url\_content **=** request**.**content

csv\_file **=** open("b\_download.csv", 'wb')

csv\_file**.**write(url\_content)

csv\_file**.**close()

I

df\_1 **=** spark**.**read**.**format("csv")**.**option("header", "true")**.**load("a\_download.csv")

df\_2 **=** spark**.**read**.**format("csv")**.**option("header", "true")**.**load("b\_download.csv")

new\_df **=** df\_1**.**union(df\_2)

new\_df**.**write**.**csv("./joined\_data")

Code 6. The source code of the Jupyter Notebook service.

Jupyter will serve as our primary platform for doing all of the required operations on the data we have obtained from Alice and Bob. In order 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 in a cluster mode.

We will then need to download the data from the other containers using the correct URL calls, where they can easily push the data that has already been “Soundexed”, hashed, and shuffled. As soon as we have finished downloading the data, we will be able to perform our union command using spark in both of the data sets. The data has been saved in a file, which will be sent back to Alice and Bob after the project is completed.

7.8 Experiment: Conclusion

We have now reached the conclusion of our experiment section, which demonstrated how it is feasible to do the PPRL using Soundex while utilizing Microservices Oriented Architecture. Since previously said, this is just the first milestone in our project's development, as it is only comprised of a very simple design. Despite the fact that the project's complexity does not approach the complexity of the real-world issues, we can clearly see how advantageous it is to work in this particular architecture. Adding additional microservices and adopting a new spark method will be discussed in more detail in the next and final Chapter 9, which will serve as the conclusion to this paper.

Chapter 8

Conclusion

8.1 Conclusion: Epilogue

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 as new technologies are introduced. It has been demonstrated in this paper that the Microservice-Oriented Architecture, its advantages and disadvantages are illustrated through an experiment. 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 in order to keep all of the transferring information safe and concealed inside of the microservices communication channel was also shown in this section. We've also shown how easy it is to integrate our apps with new frameworks by 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 transplant time has ended. As an added bonus, we've shown the advantages for novice developers who may be involved in the development of the apps 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 may get from this design, since this is often the first variable that organizations consider before implementing any significant changes. The use of this architecture may help industries and companies that are striving for success and maintaining a solid position in a competitive market, thus avoiding losses that might otherwise jeopardize the integrity of the enterprise.

8.2 Conclusion: My Final Thoughts

Since I was a novice programmer and developer, I had to deal with the majority of the problems that monolithic-based applications bring. There were no significant issues with the time commitment I made to this experiment in order 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 article concentrates. After everything is said and done, more time has been spent learning the framework's application rather than learning the application of MOA, which serves as a compelling justification for continuing to adopt this approach in the future.

Chapter 9

Future Enhancements and Improvements

9.1 Updates

In Chapter 9, which will also be the last chapter, we will discuss some of the ways in which our project might be improved. It is essential to note, however, that although the majority of these suggestions will be presented in principle, many of the ideas have already been partly implemented in reality. The design pattern using these suggestions is shown in the following picture 10.

Diagram

Description automatically generated

Figure 10

The installation of a Spark Cluster service is one of the most significant, if not the most important, of these. Furthermore, it is intended to operate in conjunction with the Apace Spark Stream libraries, and we might use it to create a real-time PPRL matching system using these.

In addition, we have linked all of the microservices to the Spark Cluster, allowing them to utilize it in the most suitable way for their specific tasks. Aside from that, we've created a Jupyter notebook as the primary interface via which we can carry out all of these tasks.

In the open repository of this experiment [30], which is shown in Figure 11, we provide the revised "open-compose.yml" file, as well as part of the code that was used to create it.

We have added three new services to our existing offerings. Cluster-c, spark-master, and spark-worker are three different types of sparks. Cluster-C was established with the intention of taking over the role of Carol. The Jupyter Notebook will be used only for the interface; rather than serving as a platform, we will simply invoke the actions and request that the services do their duties.

At long last, there are two services left to be installed: the spark-master node and the spark-worker node, both of which will be needed to execute Spark Stream operations on many nodes in a distributed system.

cluster-c:

container\_name: cluster-c

build:

context: ./services/compiler-service

dockerfile: Dockerfile

ports:

- "9400:9400"

environment:

- PORT=9400

depends\_on:

- cluster-a

- cluster-b

environment:

- PORT=9400

volumes:

- cluster-C-volume:/var/lib/data

spark-master:

image: bde2020/spark-master:3.1.1-hadoop3.2

container\_name: spark-master

ports:

- "8080:8080"

- "7077:7077"

environment:

- INIT\_DAEMON\_STEP=setup\_spark

volumes:

- shared-workspace:/opt/workspace

spark-worker-1:

image: bde2020/spark-worker:3.1.1-hadoop3.2

container\_name: spark-worker-1

depends\_on:

- spark-master

ports:

- "8081:8081"

environment:

- "SPARK\_MASTER=spark://spark-master:7077"

volumes:

- shared-workspace:/opt/workspace

END OF PAPER

BIBLIOGRAPHY AND REFRENCES

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<https://solace.com/blog/microservices-advantages-and-disadvantages/>
2. Docker Inc , 2021, https://docs.docker.com/
3. <https://en.wikipedia.org/wiki/Docker_(software)>
4. Docker Inc , 2021, <https://docs.docker.com/get-started/overview/>
5. Docker Inc , 2021, <https://docs.docker.com/docker-hub/>
6. Docker Inc , 2021<https://docs.docker.com/storage/volumes/>
7. Docker Inc , 2021<https://docs.docker.com/compose/>
8. <https://aws.amazon.com/big-data/what-is-spark/>
9. 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).
10. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation), 2021,<https://en.wikipedia.org/wiki/Lazy_evaluation>
11. Odell, M., Russell, R.C.: The soundex coding system. US Patents, 1261167 (1918)
12. Karakasidis A. ,Koloniari G. : Phonetics-Based Parallel Privacy Preserving Record Linkage , University of Macedonia, Thessaloniki, Greece
13. Karakasidis A ,Verykios S. V: Privacy Preserving Record Linkage Using Phonetic Codes, Department of Computer and Communication Engineering University of Thessaly Volos, Greece

1. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation" \o "Wikimedia Foundation), 2021,https://en.wikipedia.org/wiki/Guido\_van\_Rossum

1. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation" \o "Wikimedia Foundation), 2021, https://en.wikipedia.org/wiki/Python\_Software\_Foundation
2. Official Site of Python <https://www.python.org/doc/essays/blurb/>
3. Wes McKinney, pandas: a Foundational Python Library for Data Analysis and Statistics
4. <https://docs.python-requests.org/en/latest/>
5. <https://github.com/jamesturk/jellyfish/blob/main/README.rst>
6. Docker Inc , 2021, [docker images | Docker Documentation](https://docs.docker.com/engine/reference/commandline/images/)
7. Docker Inc , 2021, [docker container | Docker Documentation](https://docs.docker.com/engine/reference/commandline/container/)

1. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation" \o "Wikimedia Foundation), 2021, <https://en.wikipedia.org/wiki/Memoization>
2. Christen, P.: A comparison of personal name matching: techniques and practical issues. In: Workshop on Mining Complex Data, held at IEEE ICDM 2006, Hong Kong (2006)

1. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation" \o "Wikimedia Foundation), 2021, <https://en.wikipedia.org/wiki/Python_(programming_language)>

1. [Wikimedia Foundation](https://en.wikipedia.org/wiki/Wikimedia_Foundation" \o "Wikimedia Foundation), 2021,https://en.wikipedia.org/wiki/Armin\_Ronacher
2. Offical Pocoo site, <https://www.pocoo.org/>
3. Official Flask Documentation, 2021, <https://flask.palletsprojects.com/en/2.0.x/>
4. Official Jellyfish Site, 2021, <https://jellyfish.readthedocs.io/en/latest/>
5. The Official Repository Code, https://github.com/kostasrazgkelis/myThesis
6. <https://github.com/kostasrazgkelis/myThesis/blob/main/docker-compose.yml>
7. <https://github.com/kostasrazgkelis/myThesis/blob/main/services/data-service-1/Dockerfile>

1. <https://github.com/kostasrazgkelis/myThesis/blob/main/services/data-service-2/Dockerfile>
2. Sarah Diguer,https://solace.com/blog/microservices-advantages-and-disadvantages/ ,2020
3. https://okeanos-knossos.grnet.gr/home/