POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Politecnico
di Torino

Master's Degree Thesis

Privacy Preserving Data Mining: a distributed approach to data anonymization

Supervisors

Prof. Paolo GARZA

Ing. Paolo PLATTER

Candidate

Paolo ALBERTO

December 2021

Summary

With an increasing number of real world applications of Data Science algorithms, the concept of data privacy and protection of sensible information has become an increasingly debated topic. This is especially true when we look at the direction taken by European legislation when it comes to data protection of EU citizens. While there are already some software solutions available on the market for algorithms that perform data anonymization, none of them are well suited for Big Data applications. In this project we propose a distributed computing approach to data anonymization, leveraging the Apache Spark engine in order to perform privacy preserving algorithms inside of a large-scale data processing environment. We will also explore the topic of data classification, with the goal of predicting the appropriate level of privacy when new data gets uploaded to the system. The final product will be a software library, capable of querying multiple data sources and applying the required algorithms to the result. This computations will be performed with two main goals in mind: protecting sensible data of individuals, while at the same time preserving as much information as possible for analysts and data scientists to work with.

Acknowledgements

I would like to thank Agile Lab S.r.l. for providing me with their expertise and resources in the field of Big Data technologies. None of this would have been possible without your help.

I would like to acknowledge the company supervisor, Paolo Platter, and all of my colleagues in Agile Lab for welcoming me and supporting me during my internship and the thesis work. A special thanks goes to everyone in the company that helped me by working side by side to the project, or via software mentorship. In chronological order: Lorenzo Pirazzini, Marco Odore, Antonio Murgia, Pietro Brunetti, Vibhavari Bellutagi, Peppo Lorusso, Giuseppe Piccolo.
I would also like to thank Stefano Principato, for choosing to develop his thesis work with me.

I wish to show my appreciation to my academic supervisor, Paolo Garza, for his availability and his precious advice, during the course of my internship as well as my thesis project.

I would also like to express my gratitude to all of my colleagues at Politecnico di Torino, that shared this journey with me during these years.

Last but not least, I would like to thank my family, for the support that they gave me during the course of my academic career.

Table of Contents

List of Tables

List of Figures

Acronyms

GDPR
General Data Protection Regulation

PPDM
Privacy Preserving Data Mining

PQP
Privacy Query Platform

PII
Personal Information Identifier

QI
Quasi Indentifier

SA
Sensitive Attributes

NSA
Non Sensitive Attributes

JDBC
Java DataBase Connectivity

Chapter 1

Introduction

In recent years the protection of sensible information has become a more and more debated topic, with the importance of data intensive processes increasing by a lot in different industries and a growing application of Data Science algorithms in business intelligence pipelines. Society has developed an increasing demand for information and most daily actions of people are recorded in a database somewhere. This information is often sold, exchanged or shared, with the risk of exposing improperly some sensible data while doing so. Data privacy concerns gained a lot of traction, with governments stepping in to protect the privacy rights of their citizens by introducing new regulations. In European legislation, the most important regulation on this topic is the General Data Protection Regulation (GDPR), which has the goal of strengthening the protection of sensible information belonging to individual citizens, harmonising data privacy laws among all EU countries.

Looking at the direction of European regulations regarding data protection, it is evident that the preservation of privacy is going to become more and more relevant when it comes to the creation of business intelligence pipelines that need to leverage sensitive information in order to gain insights and create value for companies.
One of the most dangerous assumptions in this field of work is the concept that removing explicit identifiers such as name, surname, phone number, is enough for the data to be shared freely. Unfortunately, as proved by Latanya Sweeney in a 1997 paper [1], this precaution alone is not a sufficient condition to ensure the impossibility of re-identification of a particular subject. In many cases, some details like zip codes, birth date, gender or other similar identifiers can be used in combination with other publicly available information to track down individuals. This could, as a consequence, expose the connection between these individuals and some sensitive attributes that should be protected by privacy according to the regulations.

The idea behind this work is to create a software library capable of performing anonymization on a data source. The main goal of the final project will be to protect sensible data belonging to single individuals, while preserving as much information as possible for analysts and data scientists to work with. There will be a trade off between the two: the user of the anonymization software will be responsible for tuning parameters, removing information in exchange for a more private data set or vice versa.
In addition to this work, we will explore the topic of data classification in relation to privacy, analyzing how predictions made with machine learning algorithms can be used to forecast the different sensitivity level for the columns of a table.

1.1 Privacy preserving data mining

We will start by giving a definition of Privacy Preserving Data Mining, that we will refer to as **PPDM** from now on.

PPDM is a broad term, that includes a series of techniques and processes capable of extracting relevant information from data sources, with the final goal of protecting the privacy of individual subjects while keeping as much useful information as possible.
If the goal was simply to hide sensible data, the task would be actually pretty simple. The challenge comes when we want to maintain some information that could be still be considered valuable from the business intelligence perspective. In the next chapters we will expand on these techniques, but first we need to lay down some definitions for the different types of attributes that we will encounter and their characteristics.

1.2 Identifiers

Let's now expand on the concept in a more detailed way: when talking about preserving the privacy of a database table, we cannot treat every column of a table in the same way. Some attributes will be considered more sensitive than others, so there should be a way to assign a different importance to them. For example, the field *social security number* will be a personal, non-disclosable information, while the attribute *age* could have less of an impact on the direct identification of a subject.
From now on we will refer to these attributes with the term **identifiers**. They will be divided into categories, based on the information that could be extracted from them by an hypothetical attacker, whose purpose is to trace back an entry in the data set to the original identity of the subject. We can highlight four different

types of identifiers.

1.2.1 PII personal information identifiers

A personal information identifier is a piece of information related to an identifiable person, which is defined as a person at risk of being identified, both directly and indirectly, by using some personal identification code, or to one or more elements that are particular to that individual's identity. We will expand on this by analyzing the definition of *natural person* given by the GDPR in the next chapters.
Typically, the goal of an anonymization process is to prevent attackers from linking one or more user's PIIs to some sensible attributes (like for example linking name and surname to private medical information) that the user does not want to share.

A more formal definition of PIIs was given in a 2012 paper by Gina Stevens [2], where personal information identifiers were defined as information related to an individual, along with all the information that could be used to trace back to the individual's identity (for example name, address, telephone number...). Some other examples of PIIs are social security numbers or other unique identification codes issued by the government (for example in Italy the government issues the fiscal code which can be used to uniquely identify an individual).

UID Unique identifiers

A unique identifier is a particular kind of personal information identifier. It is a piece of information that is guaranteed to be unique among all other identifiers of the same kind. It is, for example, a serial number or a fiscal code. In the context of a relational database, a UID can be associated with a primary key, in order to impose a constraint on the column, thus avoiding repetition by design. It is usually suppressed completely in most anonymization processes, or at least it is heavily censored (for example replacing the majority of digits of a social security number with * symbols). A wrong assumption that was made in the past was to anonymize a database by suppressing all personal and unique identifiers, leaving the remaining part of the data in plain text. We will explore why this is a simplistic approach in the next sections.

1.2.2 QI quasi identifiers

A quasi identifier is a piece of information that, taken by itself, does not contain any detail capable of directly identifying a single individual: however, when combined with other quasi identifiers, it can lead to the creation of a unique identifier. Examples of quasi identifiers include information like zip code, age, date of birth...

So when we are working with a single quasi identifier attribute we don't have to worry since the danger of QIs comes when they are used in combination.
In a 1998 paper [3], Samarati and Sweeney define quasi identifiers as a set of table attributes whose release must be controlled.
One of the main challenges when it comes to privacy preserving data mining is that, no matter what we do, we can't predict possible leaks from data sources that are not under our control. Because of this risk we need to protect the sensitive attributes against cross-identification attacks.

Quasi identifiers assume an important role in data anonymization techniques. In fact, the most complex problem in privacy models is to hide the identity of a natural person from re identification, while at the same time preserving as much quasi identifier information as possible. We want analysts and data scientists to be able to use as much information as possible, while still preventing an attacker from identifying a natural person.
This will be a trade off and, at the end, the responsibility will fall on the final user of the data anonymization library. In fact the final user will always be the one in charge of making the choice when performing the queries: providing more privacy or giving away more information in order to work with a richer data set. We will not try to answer this question, since the answer changes according to the type of data and the needs of different users of the platform.

1.2.3 SA sensitive attributes

Sensitive attributes are details that the user does not want to disclose to the public. It means that, in a well anonymized database, it should not be possible to associate a single individual to a S.A. Examples of these attributes are: private medical information, unpaid tax liabilities, criminal record...

The presence of a sensitive attribute in the database creates the need for complete anonymization of the user. The final goal of a de-identification process would be to keep most of the sensitive attributes as plain text, in order for data scientists to create reliable prediction algorithms, while making it impossible to re identify a single individual. This will be achieved thanks to algorithms called *privacy models*, capable of keeping sensitive attributes in plain sight, while hiding the details that could possibly lead to the connection between those attributes and single individuals.

1.2.4 NSA non sensitive attributes

A non sensitive attribute is a piece of data that does not fall in any of the above mentioned categories. They provide little to none knowledge about the identity of a single subject.
Usually we can consider them as "harmless" for what concerns the risk of re identification. They will be kept as plain text for all the purposes of this project, and we will consider them at zero risk when it comes to the possibility of re identifying a single subject.

	PII	NSA	Quasi Identifiers (QIs)			SA
ID	Name	Height	Age	Zip Code	Marital Status	Crime
1	Joe	5	29	32042	Separated	Murder
2	Jill	4	20	32021	Single	Theft
3	Sue	6	24	32024	Widowed	Traffic
4	Abe	5	28	32046	Separated	Assault
5	Bob	7	25	32045	Widowed	Piracy
6	Amy	6	23	32027	Single	Indecency

Figure 1.1: A simple dataset with different types of identifiers

1.3 Anonymization techniques

When we use the term *anonymization technique*, we are talking about a procedure that consists in the application of a transformation to an input data set, with the aim of hiding partially or totally some details that are considered to be private, making it impossible to perform the re identification of single subjects or entities. The most basic technique is suppression, that removes completely a data point (which in general is the most extreme case), while a more sophisticated one (that is capable of preserving more information while still providing some form of anonymity) is *generalization*.

These procedures are the building bricks on which the actual anonymization algorithms will be built. These algorithm are called *privacy models*, and they leverage anonymization techniques like generalization, suppression and others, based on the column of the data set and the level of privacy that was required by the user at the beginning. We will talk about privacy models in the next section, but first we will describe the basic anonymization techniques on which they are built, starting from suppression.

1.3.1 Suppression

The basic idea behind the suppression technique is to remove some part, or even the whole data entries, by replacing it with characters that do not allow for re-identification. For example, a ZIP code could be suppressed from 10138 to 101**, in order to hide some information that could be used to identify a particular city area. In the most extreme case, the zip code would become completely suppressed: *****. This would happen when the algorithm of the selected privacy model is not be able to find a lower level of suppression that is still able to preserve the desired secrecy required by the user.
Usually, suppression is applied by following a specific rule and sticking to it for the entire column. In this software project, we decided to provide two separate methods: one for total suppression, capable of replacing every element of the column with a string of choice, and another for partial suppression, capable of suppressing a portion of the data by using a regular expression to select a part of the string (for example in some tests we used a regex that was able to select the last 2 characters of a string).

While basic suppression can reduce the disclosure risk significantly, it can also cause a significant loss of relevant information. In a 2019 paper [4], Orooji and Knapp proposed a novel approach to reduce re-identification while preserving data utility. They proposed a metric to assess the risk of disclosure for a single data point. On top of that, a value suppression algorithm was put in place to suppress only the records with the highest risk. This reduced the number of data points at risk by 45%, with an information loss of only 0.39%.

1.3.2 Generalization

In generalization, the goal is to define a hierarchy for each column that we want to apply generalization on. For example, in table 1.1, when dealing with the column *education type*, we can apply different levels, creating a so-called *generalization hierarchy*, by starting from level 0 (the original value of the column) and generalizing more and more at each level, until we obtain complete anonymization in the last level. In contrast to normal suppression, here we can have multiple levels of privacy,

and we could be able to reduce disclosure risk while preserving at least some details, that could significantly improve data science pipelines. In the case of numerical

Level 0	Level 1	Level 2	Level 3
Bachelors	Undergraduate	Higher education	*
Some-college	Undergraduate	Higher education	*
11th	High School	Secondary education	*
HS-grad	High School	Secondary education	*
Prof-school	Professional Education	Higher education	*
Assoc-acdm	Professional Education	Higher education	*
Assoc-voc	Professional Education	Higher education	*
9th	High School	Secondary education	*
7th-8th	High School	Secondary education	*
12th	High School	Secondary education	*
Masters	Graduate	Higher education	*
1st-4th	Primary School	Primary education	*
10th	High School	Secondary education	*
Doctorate	Graduate	Higher education	*
5th-6th	Primary School	Primary education	*
Preschool	Primary School	Primary education	*

Table 1.1: An example of categorical generalization hierarchy.

data, we could use a different approach, creating levels based on numerical ranges. In this way we don't need to list all the possible numbers, but instead we can just define a set of ranges that include all possible values for the records in question. For example, in the case of age we could define a hierarchy similar to the one in table 1.2. The values at lowest level (left of the table) belong to the ground domain of the attribute (original values). The highest level on the right corresponds to the maximum possible level of generalization achievable. Even a small generalization, like the one showed in level 1, is capable of greatly increasing the difficulty of re-identification, while still preserving useful details. This technique is explained thoroughly in a 2014 paper [5].

In this work, generalization (also called recoding), is described as a replacement of the values of a particular attribute with more generic ones, capable of still offering information, while hiding some details. In the example we are converting all the ages between 21 and 25 years old to a single common value [21, 25].

We are able to generalize further by creating new levels, until we reach a point in which we cannot generalize anymore. This final level is composed of a single possible value capable of defining every possible age. In our examples we use the * jolly character to represent all the possible values for that particular column. In

Level 0	Level 1	Level 2	Level 3
1	[1, 5]	[1, 10]	*
2	[1, 5]	[1, 10]	*
3	[1, 5]	[1, 10]	*
4	[1, 5]	[1, 10]	*
5	[1, 5]	[1, 10]	*
6	[6, 10]	[1, 10]	*
7	[6, 10]	[1, 10]	*
8	[6, 10]	[1, 10]	*
9	[6, 10]	[1, 10]	*
10	[6, 10]	[1, 10]	*
11	[11, 15]	[11, 20]	*
12	[11, 15]	[11, 20]	*
13	[11, 15]	[11, 20]	*
14	[11, 15]	[11, 20]	*
15	[11, 15]	[11, 20]	*
16	[16, 20]	[11, 20]	*
17	[16, 20]	[11, 20]	*
18	[16, 20]	[11, 20]	*
19	[16, 20]	[11, 20]	*
20	[16, 20]	[11, 20]	*
...	...	...	...

Table 1.2: An example of numerical generalization hierarchy.

the final version of our software project the user will be able to choose this jolly character, along with the whole generalization hierarchy, that will be stored in a Json object.

When we use the term *generalization hierarchy*, we are talking about a data structure that contains all the information regarding the different anonymization levels and their relationships between each other. Generalization can be performed in a global or local way. Local generalization can assign different rules to the same type of attribute, so that some instances keep their original value, while others obtain various levels of generalization. On the other hand, global generalization consists in the application of the same rule to every record of a particular column (all the values are generalized at the same level of the value generalization hierarchy).

1.4 Privacy models

We already explored some basic anonymization techniques. Now we will see a class of algorithms that are capable of determining how and when to apply these procedures in order to achieve a specific goal. We will refer to these algorithms as *privacy models*. Privacy models aim at defending a data set from some kind of attack. We can define different types of privacy models (each one targeted to protecting against a specific attack), with some metrics associated to them that can be used to define the desired level of privacy required.

The user will be able to change the value of these metrics in the configuration file. The decision of the user will be based on the trade-off between anonymization and query efficiency. Every privacy model will have its own metrics, that are going to be required as input from the final user of the library. In the final software application, we will use the concept of *rules* to apply privacy models. A rule will be just a combination of privacy model and specific values for metrics associated to the privacy model. Thanks to this abstraction the user of the library will be able to define multiple rules for the same privacy model, changing the model's parameters.

1.4.1 K-anonymity

A data set is said to have the K-anonymity property if the information for each element in the data is not distinguishable from at least K - 1 elements whose information also appear in the data.
So the basic concept is the one of *safety in groups*: it means that I am safe if I share the same information with at least K - 1 individuals. The concept was firstly introduced by Latanya Sweeney and Pierangela Samarati in a 1998 paper [3].

In this work, Sweeney and Samarati showed how K-anonimity can be enforced by generalizing and/or suppressing quasi identifiers. In addition to this, they introduced the concept of *minimal generalization*: a generalization is considered to be minimal if the data is not generalized more than necessary to obtain the K-anonymity property. In addition to this, the definition of *preferred generalization* was introduced to illustrate how an user could choose between different possible minimal generalizations, selecting the one that best suites some particular conditions, like favoring certain attributes with respect to others. As an example, in dataset 1.1, one could be more interested in preserving information about the age of patients, while not caring too much about their marital status or zip code. Preferred generalization would take this into account when performing the K-anonymization algorithm. This will prove to be computationally intensive to achieve, thus generating a series of papers that will describe different algorithms to

tackle the problem. We will talk about this literature in the next sections.

In figure 1.2 (a) we can see that, for each quasi-identifier (or combination

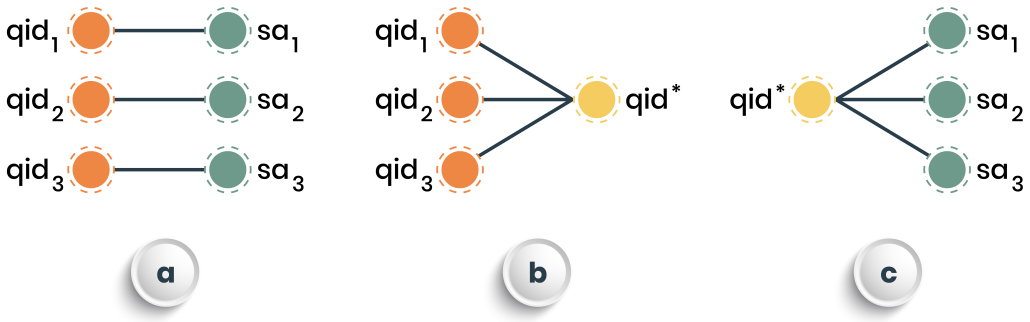

Figure 1.2: Graphical explanation of the K-anonymity algorithm

of quasi-identifiers), there is a different sensitive attribute. In order to implement anonymization, we find a new QId capable of representing all of the three original Qids. This will create a K-anonymous data set with K = 3. In general, for a dataset to be considered K-anonymous, every possible combination of quasi identifiers has to be matched to at least K individuals.

But how do we turn a data set into K-anonymous?

As highlighted in paper [6] by LeFevre and DeWitt, the challenge of finding the optimal partition when dealing with multiple quasi identifiers is considered to be a NP hard problem. However, some techniques can be adopted to find an approximate solution of the problem.

Mondrian algorithm

The Mondrian algorithm for the solution of K-anonymity problems was proposed by by Kristen LeFevre in one of his papers [6], and it uses a greedy, top-down search algorithm to partition the original data into smaller and smaller groups. If we plot the resulting partition boundaries in 2D, like shown in figure 1.3, they

resemble the pictures by Piet Mondrian, hence the name. Each group obtained, containing at least K records, is then generalized.

To our knowledge, Mondrian is the fastest algorithm for K-anonymity that is able to preserve data quality at the same time. While LeFevre provided the pseudo code in the original paper, the actual code is not publicly available. However, an open source Python implementation of the algorithm was uploaded by Qiyuan Gong to a dedicated GitHub repository [7]. The basic workflow of Mondrian is here explained:

1. Partition the data into groups of at least k records by using the kd-tree data structure.

2. Apply generalization so that each group has the same QI.

Some version of this algorithm is used in most of the widely available libraries and online tools that promise an implementation of the K-anonymity algorithm. We will talk about these tools in the next chapters, and we will exploit one of them to write a library capable of enforcing the minimum level of aggregation.

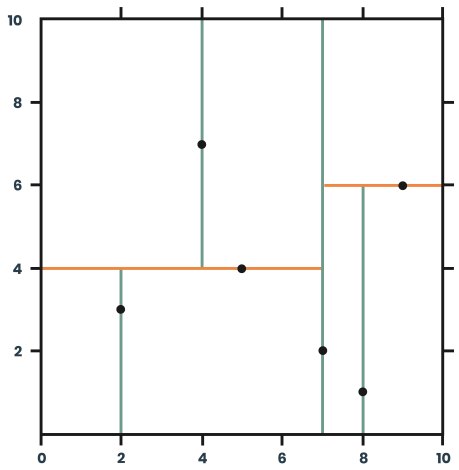

Figure 1.3: Graphical illustration of the Mondrian algorithm in a 2-dimensional space

Optimal K-anonymity

The problem of optimal K-anonymity lies in the search for a separation between attributes known to be *the best possible* one. More precisely, when suppression is allowed, we would like to find an anonymization criteria that produces the optimal K-transformed data, as determined by some cost metric. Such anonymization is said to be optimal.

In a 2005 paper [8], Bayardo and Agrawal proposed a new approach to the problem, exploring alternative strategies to reduce reliance on computationally expensive algorithms like sorting. They implemented a tree search strategy exploiting techniques such as cost-based pruning, demonstrating successful results with wide ranges of values for the parameter K. However, this is only a theoretical approach to the topic and there is not a practical project or a code repository that we could refer to. We decided to stick with algorithms that were available in a practical and already tested form for our final project.

Critical issues in the application of K-anonymity

When testing this algorithm with small data sources, one could encounter some obstacles. The first concern is related to data size. The dataset on which we are performing the test must not be small in size, or else performing K-anonymity on a great number of quasi-identifiers will only produce a small amount of rows with a really low level of information being actually shared (or possibly even one single row of completely anonymous data). This defeats one of the purposes of the algorithm, which is to preserve data quality as much as possible.

Another obstacle was encountered when testing the algorithm on actual sensible data from real world data sources: the problem is that it is difficult to find a real-life database that fits our needs, since we need private data from people. For our tests, we used a simulation engine to generate a great number of fake records.

1.4.2 L-diversity

With K-anonymity we saw how each record can be made indistinguishable from at least K - 1 records with respect to certain identifying attributes (quasi identifiers). In a 2007 paper [9], some researchers from Cornell's department of computer science showed how some specific types of attack can undermine the privacy of K-anonymized datasets.

L-diversity was proposed as a solution to prevent the problems caused by the Homogeneity Attack: this attack leverages a situation in which all the values of a

sensitive value within a set of K records are characterized by a very little diversity between them. In such cases, the sensitive value for the set of K records could be predicted exactly by the an attacker. One could think about diversity as computing entropy. In information theory, entropy is defined as the level of "surprise" for the possible outcomes of a random variable. If entropy is low it means that there is less of a surprise factor for the possible outcome of a certain random variable. If instead the entropy is high, the probability distribution is well spread and we are not able to guess easily the outcome. In this case, the diversity is high and so we can worry less about an attacker guessing a sensible attribute. This would not be the case if the distribution of sensible attributes was skewed towards a certain value.

In the above mentioned paper, the researchers showed how an attacker with background knowledge on certain target individuals could bypass the K-anonymity constraint. In table 1.3 we can see an example of a simple data set. We decided to apply a K-anonymity algorithm, with K = 4 as the only parameter and we let the algorithm run. We obtain the result shown in table 1.4. While this could seem like a good anonymization, there are still some problems in this particular example. An attacker could theoretically possess some background knowledge on a particular subject. For example, by knowing that the target subject is older than 30 and lives in a certain city, the attacker could deduce that this person has a certain medical condition (cancer). While this effect tends to reduce with an increasing size of the dataset, it is still statistically probable that we will have at least one group characterized by low diversity. Here is where the l-diversity algorithm comes into play.

L-diversity ensures that each k-anonymous group contains at least L different values of the sensitive attribute. The goal is to find new rules for suppressing and generalizing quasi identifiers such that we maximize the diversity of each group. While in small tables l-diversity often creates a great loss of information with respect to pure K-anonymity, it is capable of scaling up well with big datasets. In table 1.5 we can see the same dataset of table 1.3, but with the application of the l-diversity algorithm, where L was set to 3. As we can see, even on a small dataset like this, an attacker would not be able to guess the condition of a patient by making assumptions based on probability.

1.4.3 T-closeness

The t-closeness model is an enhancement of l-diversity. Like l-diversity, it is used to prevent the risks that could arise with the Background Knowledge attack, in which an attacker tries to find an association between quasi identifiers and the

	Quasi identifiers			Sensitive
ID	Zip Code	Age	Nationality	Condition
1	13053	28	Russian	Heart Disease
2	13068	29	American	Heart Disease
3	13068	21	Japanese	Viral Infection
4	13053	23	American	Viral Infection
5	14853	50	Indian	Cancer
6	14853	55	Russian	Heart Disease
7	14850	47	American	Viral Infection
8	14850	49	American	Viral Infection
9	13053	31	American	Cancer
10	13053	37	Indian	Cancer
11	13068	36	Japanese	Cancer
12	13068	35	American	Cancer

Table 1.3: Simple example of a dataset without anonymization.

	Quasi identifiers			Sensitive
ID	Zip Code	Age	Nationality	Condition
1	130**	<30	*	Heart Disease
2	130**	<30	*	Heart Disease
3	130**	<30	*	Viral Infection
4	130**	<30	*	Viral Infection
5	1485*	>40	*	Cancer
6	1485*	>40	*	Heart Disease
7	1485*	>40	*	Viral Infection
8	1485*	>40	*	Viral Infection
9	130**	3*	*	Cancer
10	130**	3*	*	Cancer
11	130**	3*	*	Cancer
12	130**	3*	*	Cancer

Table 1.4: Example of a 4-anonymous data set that could still be vulnerable to some attacks

sensitive attribute, with the goal of reducing the field of possibilities for the SA. For example, in [9], it was shown how knowing that heart diseases were occurring at different rates in Japanese patients could help a possible attacker to narrow down the possible values of the sensible attribute *disease*. While l-diversity treated every value of a certain attribute in the same way, disregarding its distribution, T-closeness demanded the statistical distribution of the sensitive attribute values

	Quasi identifiers			Sensitive
ID	Zip Code	Age	Nationality	Condition
1	1305*	<40	*	Heart Disease
2	1305*	<40	*	Viral Infection
3	1305*	<40	*	Cancer
4	1305*	<40	*	Cancer
5	1485*	>40	*	Cancer
6	1485*	>40	*	Heart Disease
7	1485*	>40	*	Viral Infection
8	1485*	>40	*	Viral Infection
9	1306*	<40	*	Heart Disease
10	1306*	<40	*	Viral Infection
11	1306*	<40	*	Cancer
12	1306*	<40	*	Cancer

Table 1.5: Example of a 4-anonymous data set after the application of l-diversity (L = 3)

in each K-anonymous group to be *close* to the overall distribution of that attribute over the entire dataset. The value **t**, given as input to this algorithm, is used to impose that the distribution of a certain attribute inside of an anonymized group must not differ by more than a threshold t from the overall distribution in the whole data. In paper [10] the Earth Mover metric was proposed as a measure for calculating the distance between the two distributions. This measure is based on the minimum cost required in order to transform one of the statistical distributions into another by moving distribution mass between the two.

1.4.4 KM-anonymity

KM anonymity is a version of K-anonymity that is generally considered weaker. However, it can be considered as better suited to work with data in higher dimensions. Like in the K-anonymity case, the algorithm works by considering a number n of quasi identifiers, but then it chooses to protect the data only towards attackers that have knowledge on a subset of m columns, out of the whole set of n quasi identifiers. The algorithm guarantees that every possible combination of these subset of m quasi identifiers appears at least K times in the anonymous dataset, while never taking into account the total number n of quasi identifiers. Usually m is a much lower value than n, and this algorithm is used when we are working with an high number of attributes. We will not consider KM anonymity in the practical section of this work.

1.4.5 δ-disclosure privacy

In a 2008 paper [11], Brickell and Shmatikov proposed a privacy model capable of protecting data from sensitive attribute disclosure. Sensitive attribute disclosure is a measure of how much *more* information an attacker could gain by observing sanitized QIs with respect to a maximally private data source, where sensitive attributes are completely separated from QIs. It works by enforcing a restriction on the distance between statistical distributions of sensitive attributes. However, differently from t-closeness, it uses a more strict, multiplicative definition of distance between distributions.

1.4.6 β-Likeness

β-likeness was firstly introduced by a 2012 paper [12] by Cao and Karras. This privacy model is related to t-closeness and δ-disclosure models and it can protect data from attribute disclosure risks. The goal is to overcome restrictions of the models mentioned above by reducing the maximum relative distance between distributions of sensitive attributes, also distinguishing between positive and negative information gains.

1.4.7 δ-presence

This model was defined in [13] as a method capable of protecting from membership disclosure. A dataset is defined as (delta min, delta max)-present if the probability for a single individual of a population of being contained in the dataset lies between delta min and delta max. To define this probabilities, the algorithm requires a population table, that needs to be given as input the user. This implies that the user must have a well established prior knowledge about the topic in question, since it needs to provide a statistically relevant distribution table.

1.5 What will be the focus of our project?

As we saw, there is already some literature regarding privacy preserving algorithms, as well as some open source projects. One of the main challenges, however, is the use of these algorithms when it comes to big data applications. When working with huge quantities of data, the execution times of algorithms can transform what was considered to be a fast algorithm into a time and resource consuming application. In the world of Big Data, scalability problems like this one are the norm. Instead of working locally on a big database and using a single costly machine, a popular solution is to leverage a cluster of smaller computers, with the goal of splitting the data and work on tinier, easier to manage chunks. We will refer to this technique

as *distributed computing* from now on.

Apache Spark, written in Scala, is a general purpose, distributed engine, capable of performing computations in a distributed fashion. The focus of our project will be the creation of a Spark job that is capable of performing anonymization techniques in a cluster environment. The final users of the software will be able to submit a query, our library will fetch the database, it will perform anonymization algorithms on the dataset according to some pre-defined rules and parameters, and it will return the final anonymized data as an SQL table to the user. The whole pipeline will make use of the Spark engine, thus leveraging the power of distributed computing and allowing for a faster execution, even on big quantities of data.

Another topic worth exploring is the classification of data: in this project, we used a medical dataset for data classification. We already defined what columns of the dataset contained personal information, what were classified as quasi-identifiers and so on.

When new data comes in, the idea is to use the information already in our hands, in order to classify new data coming in. To do this, the proposal is to merge the work done in the main thesis project with the fingeprinting project. This is a software project that was developed during my internship in Agile Lab. Since it was already developed during the internship, it will not be the main focus of the thesis work. We will mainly focus on the anonymization algorithms, and we will dedicate only one section of this work to the data classification algorithm.

Chapter 2

Related works

We can find a lot of work in literature that focuses on anonymization techniques. There is a good number of papers from various researchers regarding both theoretical concepts and practical projects. As for the latter, we found several open source projects and GitHub repositories developed by researchers. We analyzed all of the most relevant projects with a free license that we were able to find, to understand how to exploit some of their functionalities in our software project. In fact, we do not have to re-invent the wheel: our final goal is to create something new, while still leveraging external libraries where needed, and if some researchers already implemented privacy models and made them available to the community via an open source license it makes sense to make good use of them.

2.1 Anonymization software

We will now analyze these practical projects with the goal of understanding their potential, studying their pros and cons and how well they could suit us during the development of our privacy preserving data mining software project.

2.1.1 ARX deidentifier

ARX is an open source data anonymization tool, capable of supporting various statistical disclosure control techniques, as well as some of the most well known privacy models. It's the result of a series of papers written starting from 2012 by Fabian Prasser and Florian Kohlmayer, regarding the use of privacy models and anonymization methods in biomedical data. The functionalities of a first stable version of ARX comprehensive of multiple privacy models were described in a 2015 [14] article by Prasser and Kohlmayer. ARX is the most popular open source library for data anonymization currently available. It supports up to 50 dimensions

and millions of records. Written in Java, it is composed by a large spectrum of tools that can be used to enforce anonymity, providing three main functionalities:

1. Syntactic privacy models like K-anonymity, l-diversity, t-closeness and δ presence.
2. Different models to calculate the re-identification risk after anonymization.
3. Tools for evaluating data utility.

One of the many advantages of the ARX library is the possibility of downloading a software with a graphical user interface that allows to use all the functions of the library without writing a single line of code. For example, when trying to perform K-anonymization on the "adult" dataset, we are able to the define multiple levels of generalization, creating a hierarchy as shown before in table 1.1. After creating the hierarchy, it is sufficient to select a privacy model. In our example, we used the most common: K-anonymity. After defining K-anonymity with K = 5, we obtain the final result as shown in figure 2.1

	sex	age	race	marital-status
15	Male	57	White	Married-civ-spo...
16	Male	56	White	Married-civ-spo...
17	Male	56	White	Married-civ-spo...
18	Male	55	White	Married-civ-spo...
19	Male	54	White	Married-civ-spo...
20	Male	54	White	Married-civ-spo...
21	Male	52	White	Married-civ-spo...
22	Male	52	White	Married-civ-spo...
23	Male	51	White	Married-civ-spo...
24	Male	51	White	Married-civ-spo...
25	Male	58	Black	Married-civ-spo...
26	Male	57	Black	Married-civ-spo...
27	Male	57	Black	Married-civ-spo...
28	Male	56	Black	Married-civ-spo...
29	Male	53	Black	Married-civ-spo...
30	Male	51	Black	Married-civ-spo...
31	Female	58	White	Never-married
32	Female	54	White	Never-married
33	Female	54	White	Never-married
34	Female	53	White	Never-married
35	Female	52	White	Divorced
36	Female	52	White	Married-spouse-...
37	Female	51	White	Divorced

	sex	age	race	marital-status
15	Male	[51, 60]	White	Spouse present
16	Male	[51, 60]	White	Spouse present
17	Male	[51, 60]	White	Spouse present
18	Male	[51, 60]	White	Spouse present
19	Male	[51, 60]	White	Spouse present
20	Male	[51, 60]	White	Spouse present
21	Male	[51, 60]	White	Spouse present
22	Male	[51, 60]	White	Spouse present
23	Male	[51, 60]	White	Spouse present
24	Male	[51, 60]	White	Spouse present
25	Male	[51, 60]	Black	Spouse present
26	Male	[51, 60]	Black	Spouse present
27	Male	[51, 60]	Black	Spouse present
28	Male	[51, 60]	Black	Spouse present
29	Male	[51, 60]	Black	Spouse present
30	Male	[51, 60]	Black	Spouse present
31	Female	[51, 60]	White	Spouse not pres...
32	Female	[51, 60]	White	Spouse not pres...
33	Female	[51, 60]	White	Spouse not pres...
34	Female	[51, 60]	White	Spouse not pres...
35	Female	[51, 60]	White	Spouse not pres...
36	Female	[51, 60]	White	Spouse not pres...
37	Female	[51, 60]	White	Spouse not pres...

Figure 2.1: Example of a K-anonymized dataset in the ARX user interface

As we can see in the image, it is now impossible to find a group of unique records containing less than 5 elements.

ARX is not only a basket containing different separate tools, but a fully-fledged software. It means that all the functions have been integrated in order to work well with each other. Since the user needs to tweak parameters in order to balance a

trade off between anonymization and data utility, ARX provides a great level of configuration, with a good number of functions dedicated to the exploration of the solution space and risk calculation.

We can define various parameters before performing the actual anonymization of the dataset:

1. **Suppression limit**: the maximal number of records that can be removed from the input dataset.
2. **Approximate**: can be enabled to compute an approximate solution with potentially significantly reduced execution times.
3. **Coding model**: Some quality models also support specifying whether generalization or suppression should be preferred when transforming data.

Most models support weights that can be assigned to attributes to specify their importance. In addition to this it is possible to explore in depth the workings of the algorithm, by plotting the lattice structure used to make decisions. There are also many useful functions that can be used to verify the risk of data before and after anonymization, like a graphical tool capable of visualizing the risk for various types of attacks. In image 2.2 we can see the visual tool provided by the ARX graphical user interface to assess the risk of re-identification for different types of attack models. The risk of re-identification can be calculated also for single quasi-identifiers. Another useful feature of the library is the Classification performance calculator. This view can be used to configure classification models and their parameters, comparing the performance of these models trained on input vs the same model trained on anonymized output data.

When it comes down to the choice between the existing open source libraries for data anonymization to integrate with our code, ARX seems to be a good compromise, since we will be writing our code in Scala, which runs on JVM, and so it can be seamlessly integrated with the Java code of ARX. Another advantage of ARX is that it is the most complete from the point of view of supported algorithms and privacy models. On top of that, we are able to use the graphical tool provided by the ARX developers to perform tests on sample datasets. This allows us to visualize the workings of the algorithms before actually writing the code.

2.1.2 Amnesia

Amnesia is an anonymization software written in Java and Javascript, that can be used for performing anonymization of personal data locally. It has a more simple approach to the application of privacy models, providing an interface capable of

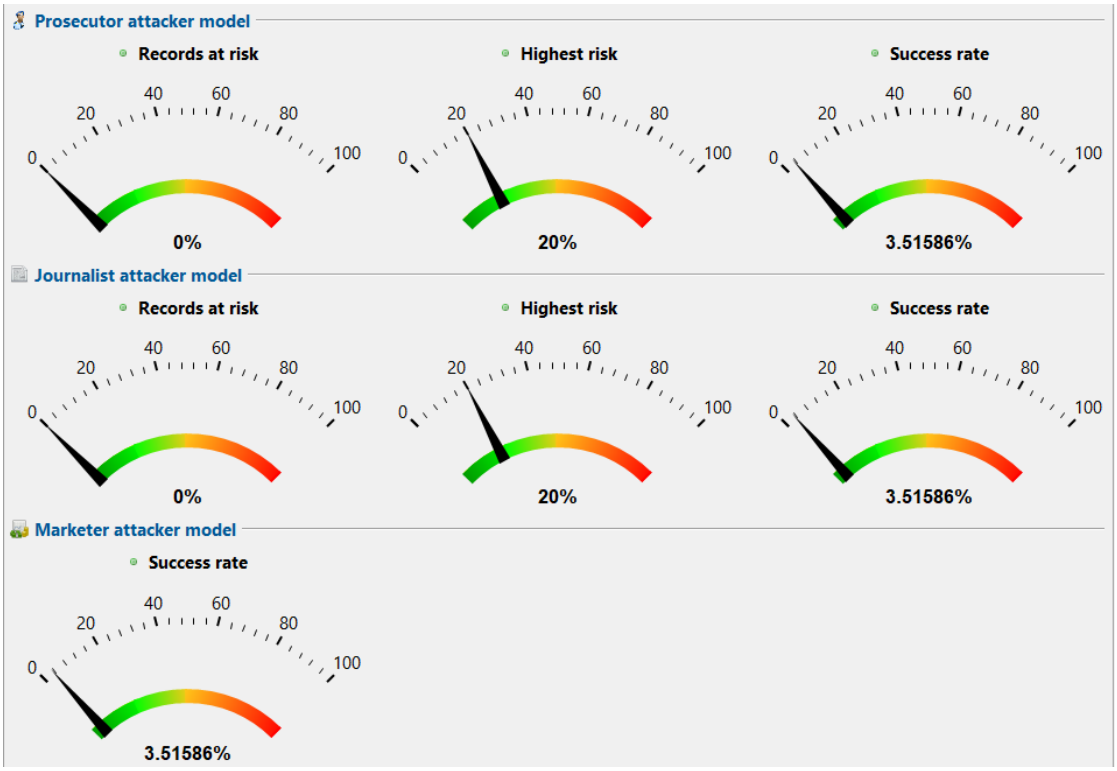

Figure 2.2: Example of a risk analysis in the ARX user interface

guiding the users trough the process of applying privacy-preserving algorithms. The goal of this tool was to create a user-friendly, free platform in order to satisfy GDPR guidelines, providing usability and flexibility to the final user. The problem with amnesia is that there is a limited number of supported privacy models to work with. In fact, only K-anonimity and KM-anonimity are supported. For this reason, we did not elaborate too much on this tool.

2.2 GDPR and anonymization

The importance of setting limitations when it comes to the disclosure of sensible attributes has grown in recent years, mostly because of the increased number of regulations. In the context of the European union, the most important regulation in the field of privacy and personal data protection in recent years has been the General Data Protection Regulation (GDPR).
The GDPR was adopted for the first time in 2016, with the goal of ensuring data protection and privacy of individuals inside of the European union and the European economic area (EEA). In addition to this it regulates the flow of personal

data going outside of the European area.

When we talk about the protection of sensible information in the context of GDPR it is important to define the concept of **natural person**. A natural person is defined in the article 4 of GDPR [15] as following:
"An identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person."

Any information related to a natural person is defined, according to the GDPR, as **personal data**. According to the text of the regulation, well anonymized data should not fall into GDPR rules at all. In fact, as we can read in recital 26:
"The principles of data protection should not apply to anonymous information, namely information which does not relate to an identified or identifiable natural person or to personal data rendered anonymous in such a manner that the data subject is not or no longer identifiable. This Regulation does not therefore concern the processing of such anonymous information, including for statistical or research purposes."

One could argue that one of the main goals of our project should be to make sure that every output coming out of the software pipeline does not fall into the scope of GDPR. However, while it could seem a straightforward task, it is not so simple to achieve a level of anonymization that is compliant with what was stated in the same recital:
"To determine whether a natural person is identifiable, account should be taken of all the means reasonably likely to be used, such as singling out, either by the controller or by another person to identify the natural person directly or indirectly. To ascertain whether means are reasonably likely to be used to identify the natural person, account should be taken of all objective factors, such as the costs of and the amount of time required for identification, taking into consideration the available technology at the time of the processing and technological developments."

One of the critical points to consider is that we need to take into account a lot of factors when making sure that an external individual is not able to identify a natural person. Some of these factors are not under our control: for example there could be a leaked database containing data that, paired with our data, it could lead a potential attacker to discover sensible information about a subject.
It should be noted that data protection algorithms should consider potential attackers, taking into account all of the reasonable possibilities. According to Opinion

4/2007 on the concept of personal data, published by the working party on the protection of individuals with regard to the processing of personal data:
"In general terms, a natural person can be considered as "identified" when, within a group of persons, he or she is "distinguished" from all other members of the group. Accordingly, the natural person is "identifiable" when, although the person has not been identified yet, it is possible to do it."

So this extends the possibility to a more "indirect" identification, which means using some pieces of information in combination with other sources to identify a single subject. In addition to this, we should also assume that an adversary could potentially possess some previous knowledge about a specific subject, chosen as the victim of the attack.

2.3 Data fingerprinting

When talk about **data fingerprinting**, we refer to a particular *signature*, associated to a data column. The idea is to define a set of parameters that can be used to describe any column of a table. This list of values will be called the **fingerprint** of that column. Some example of parameters used in the fingerprinting project include: number of distinct elements found in the column, average length of a string, percentage of numeric characters...
In the fingerprinting project, developed in collaboration with Agile Lab during my internship, we decided to extract a series of metrics for every column: They are are divided in five kinds:

1. **String-wise metrics** : For these features the raw column is considered. The mapping computed are : Grams, AlphabeticChars, SpecialChars, NumericChars. For each of these mapping three metrics are calculated : Mean, Variance and Median, ending up with 11 features

2. **String-length metrics**: Mapping the strings for their length, some numerical statistics are computed (mean, variance, skewness, etc.), ending up with 16 features

3. **Charachter-wise metrics**: For each of the printable ASCII characters (96), 9 different statistics are computed. Ending up with 864 features

4. **Word embeddings (word2vec)**: Each cell of the column is mapped to the space of word embedding. The pre-trained dictionary used map each word in 50 dimensions, then the results for each cell are aggregated with 3 different statistics : Mean, Variance and Median. Ending up with 150 features.

5. **DataSketch metrics**: Metrics that can be computed by using only the statistics obtained by using the Apache Data Sketch library. This library has been designed to deal with massive quantities of data, while still providing high performance. These metrics are actually divided in 2 separated metric types:

   DataSketch metrics: a series of metrics calculated starting from the original columns

   DataSketch lenght metrics: calculated starting from the lengths of the elements contained in the input column.

In the end, for each column we were extracting almost 1000 metrics. Each one of them can become a feature in the context of a classification algorithm of choice. All those features were extracted through the Spark engine in a distributed fashion. It is possible to select only some of them for the extraction: as a matter of fact, the word2vec features were disabled by default, because of the high computational power needed to generate word embedding operations on big quantities of data. The final output was a list of floating point values for each column of the data. Each one of these values was used as a feature in a distributed random forest classification model.

2.3.1 Labelling prediction

In order to assign labels to columns at test time, a knowledge base was used as training set, a distributed random forest model was built, using the H2O platform. H2O is an open source, distributed and scalable platform that allows the creation of machine learning models in a environment oriented to big data technology. It provides an easy way to deploy the algorithms in a production environment. In fact, the final classification algorithm obtained can be dowloaded as a Java object and directly imported in a JVM project.

The distributed Random Forest (DRF) used in this project is a powerful machine learning tool. Given the training data as input, DRF generates a forest of classification trees. Each of them is a weak learner built on a subset of the original dataset. The use of multiple trees aims at reducing the variance of the final algorithm. The average prediction over all of the trees is then calculated in order to obtain a final prediction.

The knowledge base for the DRF was extracted from a MIT database built for a similar purpose. This model was saved and then used to perform the prediction. Therefore at test time, the feature extractor was used to carry out the 901 features, then those feature were sent to the H2O model that returned the probabilities of

the column belonging to each of the classes the model was trained with.

It is possible to create custom models by using the H2O flow framework, changing the training set and tweaking hyper-parameters. In addition to this, we can also to change the number of input features. However, this needs to be well documented, since the user will need to provide the same set features in the prediction process. The results in terms of probabilities returned from the model are then combined with the data obtained through the percentage of elements of the column matching some specific regular expressions. The rationale behind this choice is that some kinds of data are easier to recognize with regex matching (e. g. email,date, etc.). For the final fingerprinting project, we decided to use various databases as training set, containing the following information:

1. Address
2. Age
3. Birth Place
4. City
5. Country
6. Date
7. Day
8. Isbn
9. Measurement
10. Name
11. Nationality
12. Nosql IDs
13. Phone number
14. Religion
15. Sex
16. Sql IDs
17. TimeStamp

One of the points in favor of the fingerprinting library is that we can train as much models as we want, we just need the input data to create a new custom model, which is independent from the other and capable of being used to make predictions on new unseen data.

2.3.2 Usage of the fingerprinting project in the context of the PQP library

In the context of our data anonymization library, the final goal would be to use the fingerprinting tool in order to assign a label to new, unclassified data. The main idea is to learn from existing data, already present in the data sources, in order to assign a label to new, never seen before, data. The goal could be to distinguish personal information identifiers, quasi identifiers and sensitive attributes. In the original fingerprinting project we used identifiers such as address, name, sex, religion... this work should not be thrown away, since we could actually create a mapping between the value of the columns and the type of identifier. For example, address and name will be mapped as personal identifiers, sex would be a QI, and religion would be considered a sensible attribute. The user of the library could define this mapping based on its needs, and we wouldn't need to train a new algorithm.

The classification will be realized with a Spark job, programmed to be executed at regular intervals (once a day or after a new data source addition).The classification results will not be permanent, and they will always be available for a review and modifiable by humans, to correct possible errors due to misclassification.

In figure 3.2 we can see the role of the fingerprinting library in the context of the main project. In the next chapter we will see the complete schema of the fingerprinting linrary, that is showed in figure 3.7. The block in which we will insert the classification software will be called **batch data classifier**. As already highlighted before, the data classification task will not be the main focus of the project. The anonymization job will be our main perimeter of focus during the development of the PQP platform.

Chapter 3

Architecture

The main focus of the project is to build a platform capable of working in a distributed computing environment, that is able to perform anonymization functions and automatic classification of new data uploaded to the data sources. This platform will be called "Privacy Query Protection platform", and we will refer to it as **PQP platform** from now on. The goal of this application is to protect medical data at national level, while still allowing users to perform queries for the sake of data analysis.

While there are already some tools to perform anonymization available in the market, none of them is built around the concept of big data. When we work with data sources containing millions of entries, the classic approach of computing algorithms on a single (even if powerful) machine does not make sense anymore. Especially when we need to perform tens of millions of simple operations, it makes more sense to distribute the computations between a lot of smaller computers. Each computer will be called a node, and we refer to the whole system with the term **cluster**. This cluster will be able to schedule the same tasks on all nodes, controlling them as if it were a single system. This will be achieved thanks to Apache Spark, a popular open source engine for large-scale data processing and distributed computing. Spark is built on the idea of distributed data, splitting the computation between nodes of the cluster.

In order to achieve the final anonymized result, our engine will be inserted between the actual data source (containing the plain data in clear) and a web interface, that we will call PQP portal. From this portal the user will be able to perform SQL queries, obtaining results with some anonymization rules applied to them. The rules that needs to be applied will be decided according to the permissions of that particular user.
We will not focus on the creation of this portal. Instead, we will firstly create the actual Spark job, providing then a REST API to the developers of the user

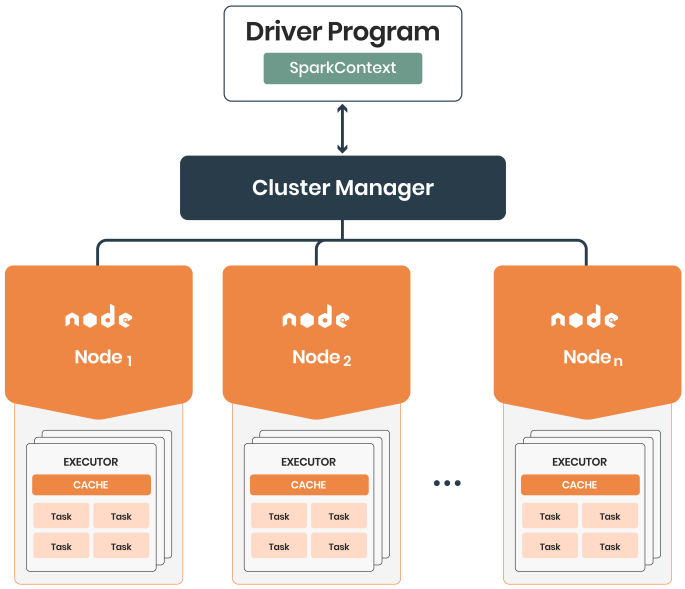

Figure 3.1: Spark components

interface. This will allow us to work on the actual core of the PQP anonymization project, building a stable API that is capable of ensuring anonymization with a focus on the safety of personal data.

The idea behind the platform is to handle a lot of users, characterized by different sets of permissions. We will call these sets **research topics**. For example, a user with a *chemistry* research topic could be allowed to access in detail information regarding chemicals in the blood of patients, but some other critical information like illnesses or medical history will not be made available to them. This will be completely configurable by the administrators of the platform, with the possibility of defining new rules for a certain research topic.
Before going into the details of the PQP platform inner workings, it is important to define who will the final users be, and what will be their roles. For the final architecture, we provided four different types of users of the platform:

1. **System administrator**: It is fundamental for handling the users of the platform and assigning roles. The SA is in charge of assigning all the other roles and changing their permissions if needed.

2. **Data Privacy Administrator**: The DPA will have the power of configuring and assigning research topics to lower-level users

3. **Data Provider**: The DP will be able to upload new datasets and will be

able to visualize the data classification, eventually changing it if needed.

4. **Data Consumer**: The final user of the platform. The DC will have a research topic assigned to them. Data consumers will provide queries to the interface, and they will receive the final query result, anonymized according to the research topic assigned to them.

One of the goals of the platform is to create an easy way for the final user to interface with the software, in order to allow for an easy integration of the final spark job in a larger data analysis platform in the future. Because of the same reason, it is required for the PQP platform to be used both on premise and on cloud.
The sintax used for the queries will be SQL, and the engine will be the one in charge of translating the SQL language to the syntax required by the underlying data source. In fact, we plan on supporting a great number of databases in the final version of the PQP platform. In the context of this project we considered 3 main data sources:

1. **PostgreSQL**: One of the most popular object-relational database management systems. Written in C, it has an history of more than 30 years of active development and a strong open source community behind. Most of the features required by the SQL standard are supported, with an history of reliability and data integrity.

2. **MongoDB**: It is one of the most popular solutions when it comes to non-relational database solutions. It relies on a document based structure, with a JSON-like schema. It is capable, in some types of applications, of outperforming traditional RDMS software, with an easier and faster data integration process.

3. **ElasticSearch**: A popular distributed search engine developed in Java, with a focus on scalability and very high search speeds.

The final user will be able to use the same language to perform queries on all these data source indistinctly. In order to achieve this we will be leveraging an SQL query engine with a focus on performance when it comes to big data applications.

Unit tests were written to make sure that all the modules contained in the final software were working correctly. These are automated tests written to make sure that a certain section of the code (called "unit") behaves as intended by the developers, failing when results are not as expected.
A CI/CD (continuous integration / continuous deployment) pipeline was deployed to perform automatic checks on the project repository, running all the tests at each

new commit. The use of a similar pipeline is considered to be a good practice in software development, since every change in the code results in a complete checkup of all the functionalities of the library. This process of continuous compilation, testing, and deployment was achieved thanks to the use of the **sbt** tool for Scala (abbreviation for "simple build tool").

We want to emphasize that the main requirement of this project was not the connection of the library to a BI tool. Instead, the efforts were mainly focused on the creation of a safe and functional data extraction tool, capable of performing anonymization tasks and protecting sensible information from leaking.
Another important point was the preservation of schema and data types, even after the anonymization. As an example, if we generalize every age from 20 to 29 years old to the string "2*", we still want to be able to know that originally the column *age* was an integer and not a string.

3.1 High level design

In image 3.2 we can see the final design of the PQP platform. The execution engine for the main job of the platform will be, as already mentioned, Apache Spark. This will allow for a great horizontal scalability and a great flexibility when it comes to the number of nodes in the cluster.

We will be writing the code in Scala language. This will guarantee a great compatibility with Java libraries, since we are working on a JVM based language, that can be fully compiled to Java bytecode. In order to create a better interaction on the user side, Apache Spark will be used in its SparkSQL interface, to facilitate the process from the user perspective. For a better interaction with the core job execution engine we will be leveraging Apache Livy, a service that allows to activate multiple Spark Sessions and to interact with them thanks to a REST API. Furthermore, from version 0.6.0 Livy provides a JDBC server. We will now analyze the architecture in its different components, starting from the metadata repository.

3.1.1 Metadata repository

In our implementation, the metadata repository was a PostgreSQL database, containing all the metadata information needed by the anonymization process. Each table contained in our datasources has its own ID, name and description, and all of its columns are declared in this database with its corresponding datatype. In addition to this, there is a relation with the underlying data source (PostgreSQL, MongoDB or Elasticsearch) via the IDDataSource field. Basically all the external information needed by the code (except for the actual data) is stored here.
In the metadata repository we also store the information on what rules needs to

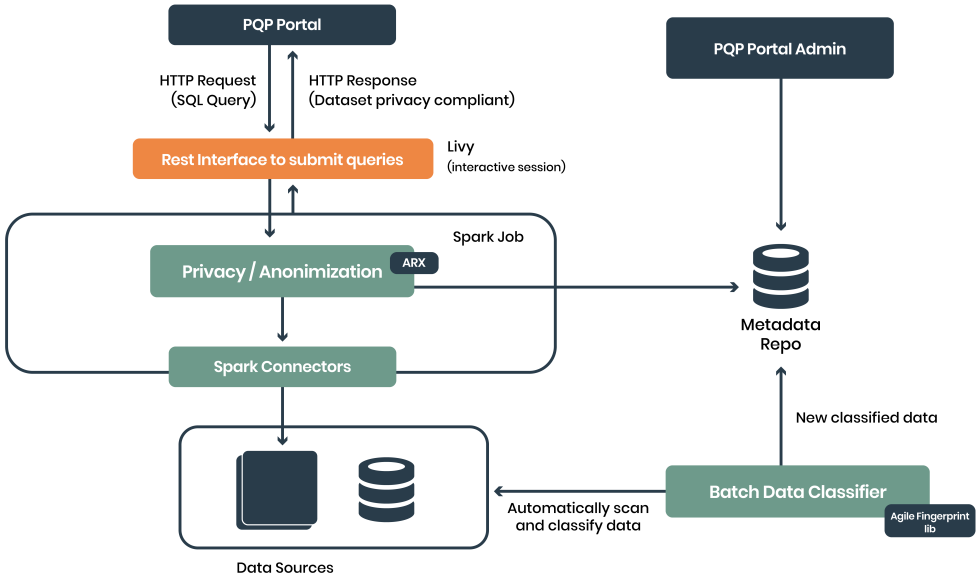

Figure 3.2: High level design of the PQP platform

be applied for a certain combination of dataset and research topic (that we will call "scope" in our implementation). Each combination of idDataset and idScope could potentially have an anonymization rule associated. The goal of the final end to end job will be to take the query as input, retrieving the rule needed by the current scope with a query on the metadata repository, applying the required rule and finally returning the anonymized output to the user.
We have to remember that some rules could be applied only to certain columns of the dataset. For example we could want to suppress only email addresses, while keeping less sensitive information as plain text. In order to take this into account, we will use two separate tables, called *column* and *columnrule*. In figure 3.3 we can see the complete graphical representation of the PostgreSQL schema for the metadata repository.

We can now expand in detail on the tables contained into the PostgreSQL metadata repository:

1. **DataSource** table

   idDataSource: unique key, represents the id of the data source (for example 1 = mongo, 2 = postgres...)

   name: name of the data source (mongo, postgres...)

   URI: Connection URI for acessing the database (for example: `jdbc:postgresql://postgres:5432/postgres`)

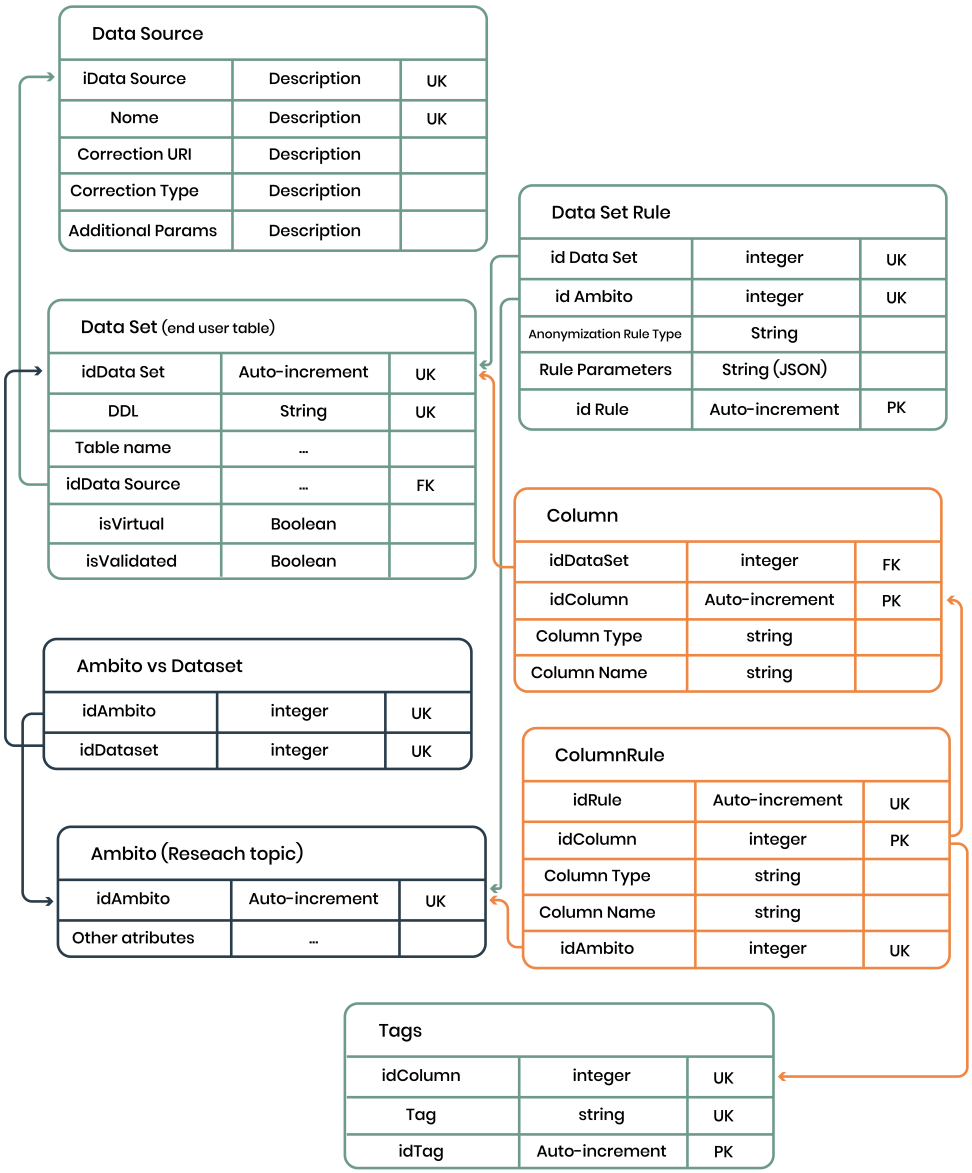

Figure 3.3: Metadata repository schema

ConnectionType: additional information on the connection to simplify the connection process on new machines(for example `PostgreSQL 14.0 (Debian 14.0-1.pgdg110+1), 64-bit`)

params: additional parameters for the connection (like username, port...)

2. **DataSet** table

   idDataSet: unique key, represents the ID of the dataset (table) in question

   DDL: Data definition language expression, in case we need to perform query virtualization

   tableName: name of the table

   idDataSource: foreign key, data source where this table is stored

   isVirtual: set to true if we need virtualization on this table

   isValid: it can be set to false if we want to block users from accessing this table completely.

3. **AmbitoVsDataset** table

   idAmbito: unique key representing the scope (ambito)

   idDataset: ID of the dataset with this particular scope

4. **DatasetRule** table

   idRule: ID of the rule

   idDataset: dataset on which this rule needs to be applied

   idAmbito: ambito (scope) for which this rule needs to be applied

   ruleType: enum that represents the type of rule to be applied

   ruleParameters: a JSON containing all the parameters needed by that specific rule

5. **ColumnRule** table

   idRule: ID of the rule

   idColumn: column on which this rule needs to be applied

   idAmbito: ambito (scope) for which this rule needs to be applied

   ruleType: enum that represents the type of rule to be applied

   ruleParameters: a JSON containing all the parameters needed by that specific rule

6. **Ambito** table

   idAmbito: ID of a particular scope

   description: description of what this scope is about

7. **Column** table

   idDataset: ID of the dataset to which this column belongs to.

   idColumn: ID of the column

   columnType: data type of the column

   columnName: name of the column

8. **Tags** table

   idColumn: ID of the column to which this tag is referred to

   Tag: tag assigned by the prediction of the machine learning model of the fingerprinting library

   idTag: id of the tag

The users of the PQP tool will not be able to see the data contained inside these tables. Only the users with administrator privileges will be able to perform read and write operations in this database. When adding a new table to one of the data sources, an automatic process will insert all the information related to the new table in the metadata repository. This repository is important not only because of the details regarding rules and their application, but also since we need a way to remember the data types of columns in each table. This information is actually really important, because anonymization processes often cause a loss on the information regarding data types. In fact as already stated before, suppressing part of an age, like for example 25 that becomes "2*", causes a change of data types, from Integer to String. The metadata repository helps us to remember that we are in fact dealing with an integer.

3.1.2 PQP Portal

This portal will be the main tool that lets the final users of the software interact with the database and perform queries. In this portal there will be a login section dedicated to the admin functionalities, accessible by users with administrative privileges. In the section dedicated to the administrator role, the logged user will be able to perform actions on the metadata repository. For example, an admin could assign a new set of anonymization rules to a particular dataset. Another privilege granted to an user with this role would be to change the permissions of particular user, by adding or removing them from a certain scope (called ambito in

our database). All these actions will be performed from a graphical web interface, where the user will be able to login and interact with all the functions of the PQP library. This application will be able to communicate with the backend thanks to the REST interface provided by the integration with Livy. As stated before, the backend was the focus of the project and we did not provide a practical implementation for the frontend part of the software tool. Because of that, all the tests were performed with POST requests using *postman*, an API platform that was used to create Livy sessions and submit statements to them.

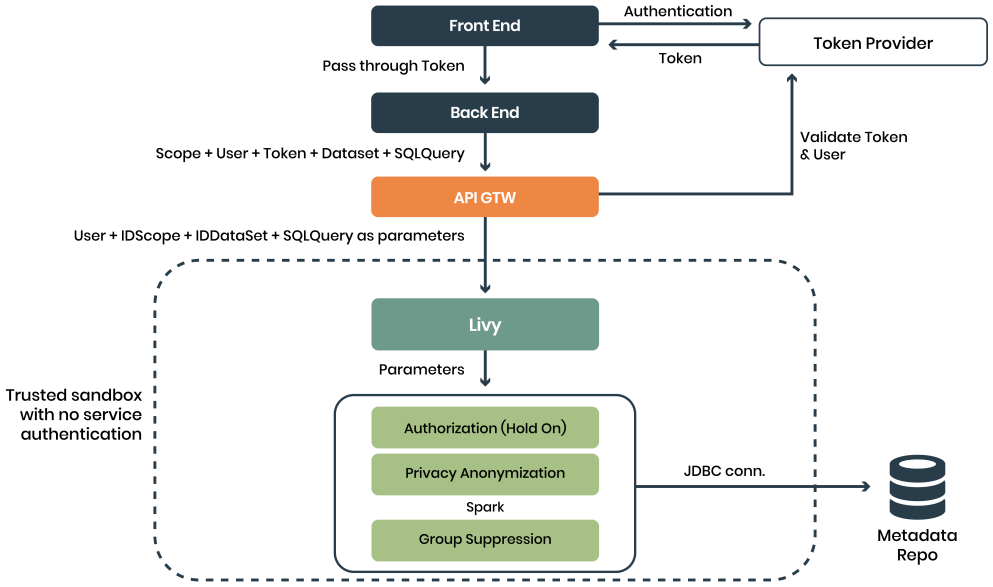

Figure 3.4: Authentication process for the PQP portal

3.1.3 Livy interface

Apache Livy enables the interaction with the actual Spark job by providing a REST interface to call functions. It enables the submission of Spark jobs as well as some code snippets, useful for testing purposes. In addition to this it can manage the Spark Context and creates generally a simple interaction between the Spark code and the servers of the application, enabling the use of distributed applications on web or mobile applications. It provides shared cache between RDDs and DataFrames across multiple jobs, to allow for faster execution. In figure 3.4 we can see details regarding the connection that Livy enables between the Spark job and the API gateway.

In figure 3.5 we can see the Livy interface with a very simple statement execution. In this example we queried a dataset (with idDataset = 1) of fake users, by selecting name, surname and age. We specified also a scope (characterized by idScope = 2). This particular combination of dataset and scope was assigned to a generalization rule, that was set to be applied only on the *age* column. As we can see from figure 3.6, details about the age of users were hidden by applying generalization.

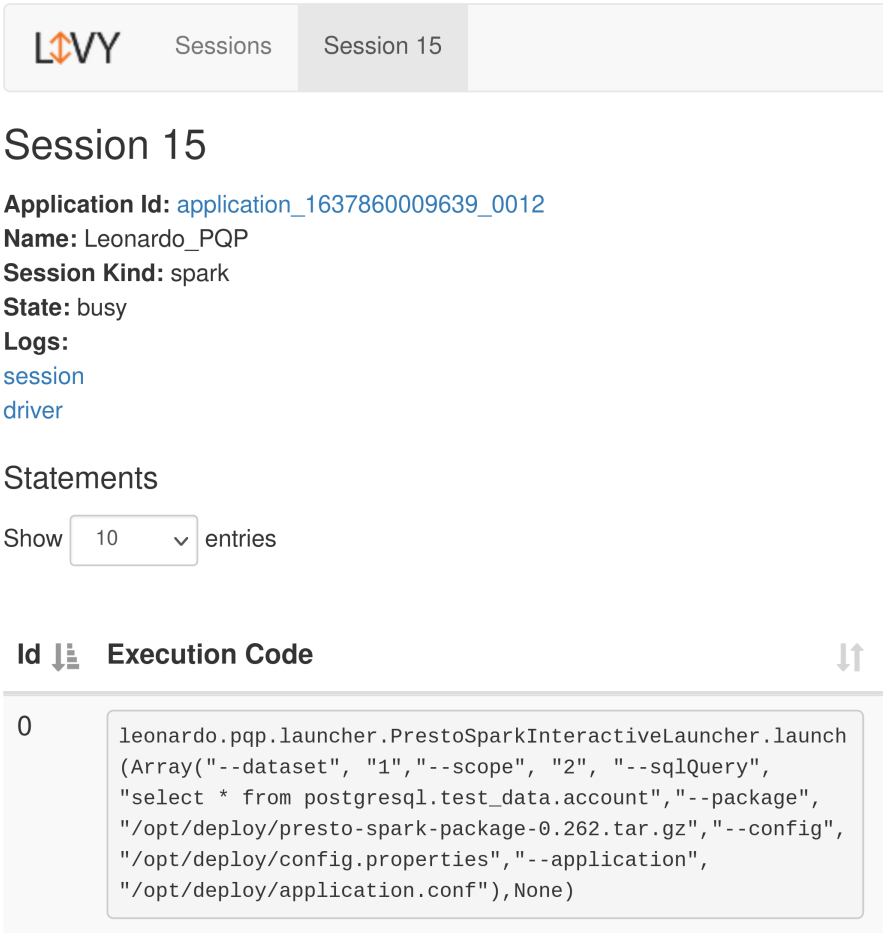

Figure 3.5: Example of a simple Livy statement

3.1.4 Spark Job

The development of the Spark job can be considered as the focal point of PQP project development phase. All the other blocks that we described in the schema were designed with the Spark distributed computing environment in mind, and

```
res2: Array[String] =
Array({"name":"Annora","surname":"Wallman","age":"0-50"},
{"name":"Aurea","surname":"La Vigne","age":"0-50"},
{"name":"Livy","surname":"Brandoni","age":"0-50"},
{"name":"Myrtice","surname":"Gallihaulk","age":"51-100"},
{"name":"Deana","surname":"Starie","age":"51-100"},
{"name":"Horst","surname":"Muggleton","age":"51-100"},
{"name":"Saunders","surname":"Pittway","age":"51-100"},
{"name":"Meriel","surname":"Terry","age":"51-100"},
{"name":"Rora","surname":"Scown","age":"0-50"},
{"name":"Victoria","surname":"Twizell","age":"0-50"},
{"name":"Boy","surname":"Semark","age":"0-50"},
{"name":"Andriana","surname":"McAllister","age":"51-100"},
{"name":"Katrina","surname":"Shevelin","age":"0-50"},
{"name":"Stephen","surname":"Fifoot","age":"0-50"},
{"name":"Fayina","surname"...
```

Figure 3.6: Output of a simple Livy statement

from the start we planned all the main architectural choices in function of this piece of software.
Let's analyze one of the main peculiarities of the Spark engine, that make it suitable for handling big data applications: at high level, every software of this kind is composed by a driver, with the task of executing the main method provided by the user, performing a series of operations in parallel. Spark provides data structures that are able to store any collection of elements across the cluster's nodes in a partitioned fashion. The most common data structure used in Spark is the RDD (Resilient Distributed Dataset). In our application we will use the Spark SQL DataFrame as our data structure of choice, since it combines the benefits of RDDs with the optimizations of the Spark SQL module, while providing a structure that is equivalent to a RDMS table. This allows for a great performance on distributed applications, coupled with a structure similar to a table in a relational database, which is exactly what we need when working with SQL statements.

It is important to define the two main types of operations that can be used to work on distributed structures inside of the Spark engine along with their key differences: transformations and actions.

Transformations

A transformation applies a change to a distributed data structure, while never stepping out from the distributed environment. For example, if we are working with an RDD, a transformation will return a new, transformed RDD, where an operation was applied to each element of the dataset. The crucial point is that transformations are evaluated lazily. This means that the data across the nodes is never actually touched, until an action is executed. It means that theoretically we could chain multiple transformations without never actually changing the content of the RDD (until, of course, an action gets executed, at which point all the transformations are evaluated one after the other). After every transformation I will get a new lazily modified RDD. Some of the most common transformations include:

1. **Map**: Applies a transformations to all the elements, one by one.
2. **Filter**: Returns only the elements satisfying a particular condition.
3. **FlatMap**: Applies a transformation to all elements creating a flattened result.

Actions

Actions are methods that actually access the data of an RDD (or another distributed data structure). Whenever an action is performed, all the transformation declared before of that action get executed at once, and the data from the different cluster nodes is extracted and gets moved outside of the distributed environment. Because of this, it is important to always make sure that the result of a transformation is capable of fitting in the local memory of the system.
Here are some common actions:

1. **Collect**: probably the most used. It collects all the elements in a single local array without any extra operation.
2. **Reduce**: it is used to aggregate the elements of an RDD according to a specific rule.
3. **ForEach**: to perform an action for each element of the RDD. For example I could use it to print all the elements.

When use the term *Spark job*, we are talking about a function that gets executed by leveraging parallel computation, consisting of a series of tasks spawned in response to a Spark action. Spark transformations alone are not enough to produce a result. There is always the need for an action at the end of the script.

Let's now talk about the Spark job that we implemented in our project. The final end to end function is capable of performing a query chosen by the user. After performing that query, the result gets anonymized according to the rules that are associated to the scope of that user.
Let's analyze the pipeline step by step, starting with the first thing that is needed by the job to launch the main function, which is the user's input:
When starting the job, there are three main inputs that the user should provide to the main function:

1. **Input Query**: written in SQL language, it is always the same, independently from the data source that we are querying. For example, if we wanted to perform a query on the user table of the mongodb data source, we could write: `select name, surname, age from mongodb.pqp.user`.
   Notice how we are performing a SQL query on a non relational database like MongoDB. This is achieved trough the use of the Presto Spark connector.
2. **Dataset ID**: the ID related to the dataset in question. Example: `1`
3. **Scope ID**: the ID related to the wanted scope. Example: `2`

The final output will be a table containing the data required by the user in the query, anonymized according to the specific rules for that particular combination of dataset and scope. In order to obtain this result, the pipeline is composed of two main blocks:

Presto connectors

The first thing to do before any transformation on the data, is to actually execute the query inside of the correct database. At this point of the pipeline we are not taking into account anonymization rules logic. The execution of the SQL statement given by the user will be performed thanks to the Presto Spark connector. Presto on Spark makes it possible to exploit Spark as an execution framework for SQL queries. This connector plays a critical role when we want to run the query on a great number of nodes, a task that would otherwise require an unacceptable quantity of memory and CPU computational power. Moreover, Spark provides some additions to the value of the codebase, like resource isolation, fine grained management of computing resources and almost immediate scalability.

Presto allows users to perform queries on multiple datasources, by using basic SQL sintax. This simplifies a lot the final usability of the BI tool, since the user will not need to change language if, for example, some tables are in the MongoDB database and some others are in PostgreSQL. We tested presto on PostgreSQL and MongoDB, and we are planning to implement ElasticSearch as a third data source in the near future.

Privacy anonymization block

Inside of this block we are performing the actual work of privacy preserving data mining. Here we leverage the ARX anonymization framework in order to apply the anonymization rules when needed. We focus our work on the anonymization functionalities of this library, leveraging the ARX algorithms inside of our big data application written in Spark. For now we will not exploit other functionalities of the arx-deidentifier software like risk assessment and other statistical functions.

Because ARX is written in Java, we have a native support, since Scala runs natively on JVM. The library has been used in a lot of various contexts, including commercial big data platforms, research projects and clinical analytic platforms. It is able to work efficiently even on commodity hardware, with a dedicated cross-platform graphical interface available on the library's website to download. We exploited this graphical interface for testing purposes, while we used the last release of the code available on the ARX GitHub repository [16] for the actual development of the PQP project.

At a software level, the anonymization block is composed of a general anonymization service trait that implements an applyRule function. For each anonymization rule supported by the final PQP application, this function has a different implementation. We will talk more specifically about the structure of the project in the next chapter, that will be dedicated to the practical implementation.

3.1.5 Batch Data Classifier

This will be a separated Spark job, based on the automatic classification of data performed by the fingerprinting library. We already talked about this application in section 2.3. Its integration inside of the main PQP platform architecture will be implemented as a Spark job, programmed to run at predefined times during the day. Another way to trigger this process will be the upload of a new table into one of the data sources. This will cause the activation of the classification engine on newly uploaded data. As an output, the job will write new entries in the *Tags* table, inside of the metadata repository, with a prediction on the type of identifier for each column of the new table. As stated before, this decision will not be permanent, but the PQP portal administrators will be able to review the result of the classification, changing them if needed. At the time of writing, the fingerprinting project is completed. However, its integration inside of the main PQP software is not. We plan on integrating the two projects together in the near future, but for now we will focus on this classification tool separately.
In figure 3.7 we can see the high level design of the library that will be used in the batch data classification process: As mentioned before, the result of the

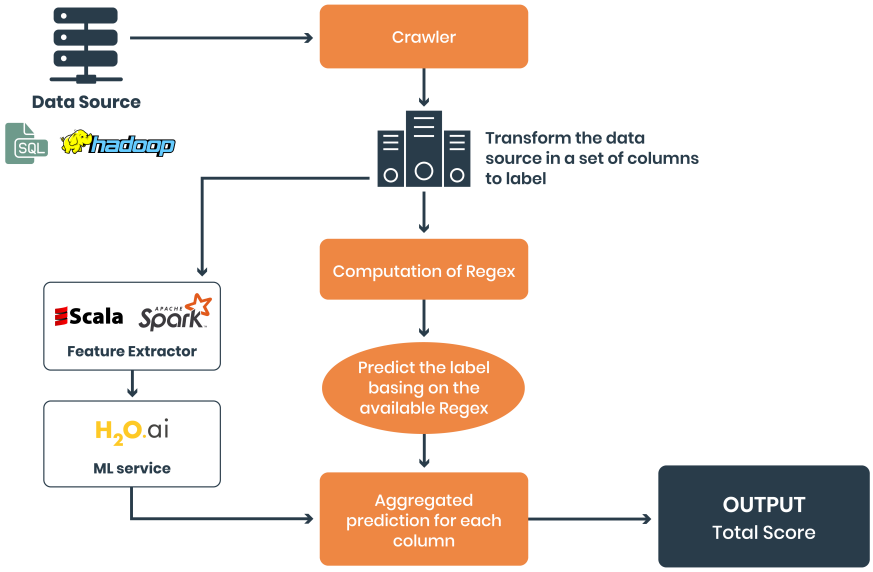

Figure 3.7: High level design of the fingerprinting data classifier

machine learning model is combined with some regex results. This is useful for some categories that have a pretty simple formatting that never changes (for example email addresses). The final output is a list of probabilities and the classification is performed by taking the category with the highest probability as the final prediction.

3.1.6 Data sources

We planned to support three data sources by using the Presto Spark connector: PostgreSQL, MongoDB and Elasticsearch. At the time of writing, only PostgreSQL and MongoDB have been actually deployed and tested with real data. These data sources were deployed on a Docker container. This allowed us to package all the required libraries and dependencies in a system capable of running in any environment, without having to set up everything from scratch when moving to a new machine. Containers simplify by a lot the delivery of distributed applications and they have become more and more popular in software development companies.

A Python script was written to create a PQP backend with the help of the Flask library. The goal of this application was to create and populate the data sources automatically, providing GET and POST methods to perform changes directly from the Linux shell.

3.2 PQP: Admin functionalities

When building the logic behind the library, an important task was deciding the set of permissions that the administrators could have. The admin should be allowed the permission of:

1. Adding new roles.
2. Performing data provider roles, like the upload of new datasets.
3. Creating new datasets by joining existing ones.
4. Reviewing metadata related to already existing datasets.
5. Reviewing the anonymization rules associated to a particular research topic.
6. Change permissions of users in order to assign different research topics to them.

During the CRUD phases, these functionalities will be enforced by leveraging the PostgreSQL schema dedicated to the metadata repository, creating methods to perform SQL statements. All of these methods will be thoroughly tested in our CI/CD pipeline.

3.3 PQP: integration with the ARX library

One of the main challenges of the PQP application development was the integration with the ARX library. Usually, in software projects where we need to use a lot of external software components, it is considered good practice to use a library repository to download and manage automatically a great number of external application imports without the risk of occurring in software conflicts.

In the case of the ARX deidentifier project, at each official release of the library the authors published a complete jar archive of the application on the dedicated GitHub repository [16], providing a version number and a short description of the new changes. The problem with this versioning method is that the only way to import the ARX library is to download the jar as an **unmanaged dependency**, putting it into a folder inside of the main Scala project. This means that interactions with the library are not hidden from the outside world and we need to write a custom script in order to download the jar from GitHub every single time we want to set up the project on a new machine.

However, this is nothing compared to another common problem of unmanaged

dependencies: the so called *Maven dependency hell* [17]. This occurs in active Java-based projects when two dependencies are declared in the same project, often with a different version number. The solution is to search for all the duplicates inside of the dependency folder, removing the ones causing conflicts. In the case of the ARX project there were some dependencies causing this exact problem, like for example Google Guava: a widely used library that is well known for causing these kinds of problems [18].

The solution to this problem was to write a custom shell script B capable of downloading only the ARX core without any of its related dependencies. In a second phase, the script downloaded ARX's correlated dependencies, excluding the ones that were causing conflicts. Finally, we simply programmed the CI/CD script to run this script at deploy time.

At a code level, we made use of ARX hierarchies and privacy models to implement some privacy rules in our project. The K-anonymity was useful for the implementation of a rule that was imposing a constraint on the *minimum level of aggregation*. For example, let's say a user of the PQP tool wants to find the average salary for each role inside of a company. The query used would be something like:

```
SELECT role, AVG(salary)
FROM employees
GROUP BY role
```

Now, everything works fine until we have a role that is covered by only one person. For example, in the case of the CEO role, the average salary of the CEO role will correspond to the real salary, causing a leak of sensitive information. To solve this problem, we introduce the concept of minimum level of aggregation, where we impose a constraint on the minimum number of records for each role. When this minimum level is not reached, the sensible data (in this case salary) gets suppressed. This goal can be achieved thanks to the K-anonymity algorithm, by setting the parameter K to the desired value. In our code we created a custom rule to impose this minimum level of aggregation and we referred to it by using the term *group suppression*.

Chapter 4

Implementation

In this chapter the different components of the project's architecture that we already talked about will be analyzed in depth, providing specific details regarding their implementation in the final project and at code level.

4.1 Metadata repository

The creation of the metadata repository was divided mainly in two phases. In the first phase we were developing the main spark job and we needed a fast way to simulate the metadata repository without actually deploying the database. This was important in the earlier stages of development, where the focus was on the creation of a very simple end to end job, without wasting energies on integration tasks. Because of this, in the first iterations of the PQP application, we used a library with the goal of simulating a PostgreSQL schema into our Scala code without external dependencies.

We chose to leverage the *Embedded Postgres* library [19], useful for unit testing purposes, since it is able to deploy real Postgres schemes automatically, without requiring the end user to actually install and setup a database cluster. To instantiate a database with Embedded Postgres we just need a couple of lines of code: first we have to set the port and then we can use a simple start() method to deploy it. After this short initialization we can instantiate a `Statement` object, used for the execution of static SQL statements and for returning their results.

Thanks to this tool we were able to focus our work on the actual code writing phase. Even when the code base was completed and we started using a real PostgreSQL database, we still exploited the Embedded Postgres dependency inside of our unit tests. This can be considered as a good practice, since it is important

to decouple the data used for testing purposes from the actual database, that will be used only in a real production environment. This is widely considered to be a good choice both from a privacy standpoint (we work with fake data instead of the real production database) and from the point of view of unit testing best practices (in unit tests we want to test single software modules, removing external influence where possible).

In the second phase of the development, after we obtained a working first version of the end to end PQP Spark job, we finally implemented a real database, with the goal of storing the actual production data in it. The first task was the creation of a Docker container, where we installed PostgreSQL. Then, we used a Python script that was leveraging the Flask library to provide a REST API for manipulating SQL tables with some simple POST and PUT methods for each table of the metadata repository schema. After that, we used this REST API to write a shell script capable of creating and populating tables.

We used the DBeaver software to explore the database, creating and testing queries. In figure 4.1 we can see the final schema of the metadata repository that we decided to use in the last version of our application.
Finally, we wrote a series of queries needed by our code. For example, we created a query that, given as input idDataset, idScope and ruleType, was used to retrieve the name of the of the columns on which the rule needed to be applied, together with the Json containing all the rule parameters required by the ARX algorithm:

```
SELECT column_name, rule_parameters
FROM column c, columnrule r
WHERE r.anonymization_rule_type = ${ruleType}
AND c.id_dataset = ${idDataset}
AND c.id_column IN
(SELECT id_column FROM columnrule r WHERE r.id_ambito = ${idScope});
```

All of the queries required by the algorithm were integrated in specific methods inside of a dedicated Scala trait. The role of this functions was not only to retrieve the information from postgres, but also to transform the raw data coming from the database into custom data structures written specifically for this use case. We decided to make use of Scala case classes for this purpose.
For example, in the case of the *suppression* rule, we created a custom case class to store all the information required by the schema (like column names, type of suppression, default string to use when suppressing...), as shown in appendix A. This data structure was used to obtain a more elegant code base, creating an application that was easier to maintain and update with new functionalities when

needed.

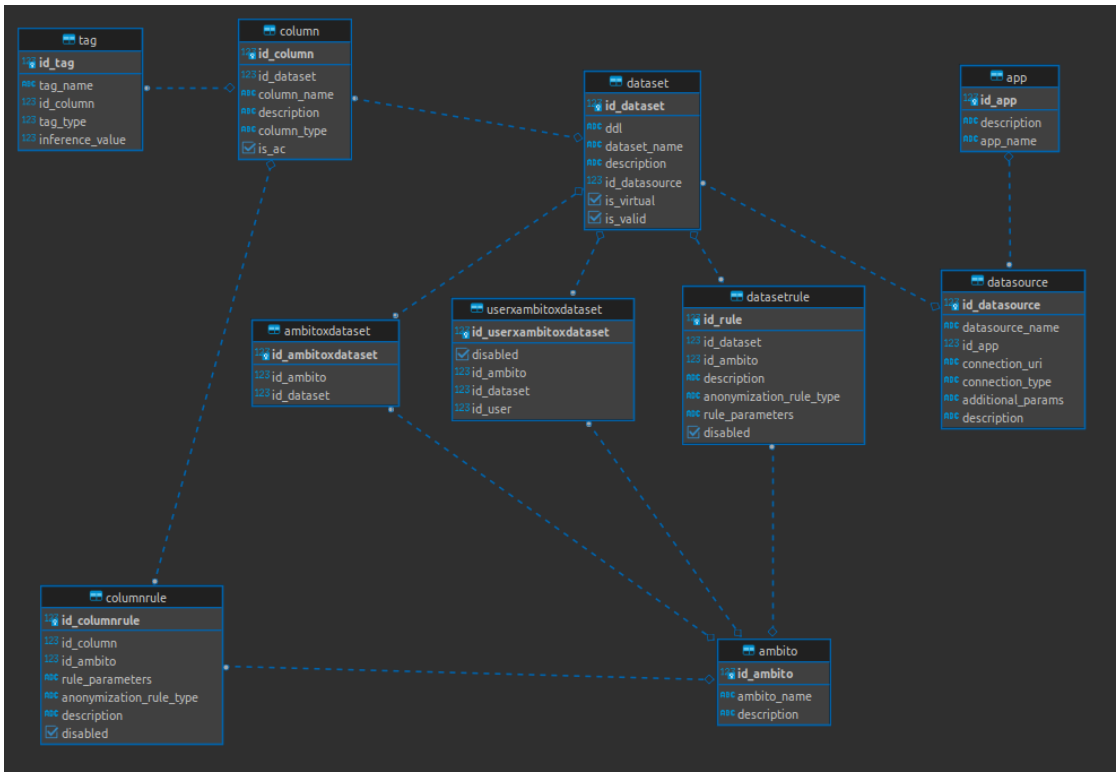

Figure 4.1: Final implementation of the metadata repository in the postgreSQL database.

4.2 Livy interface

As explained in the previous chapters, Livy enables the submission of Spark jobs from external sources like web interfaces, smartphone apps and so on. Multiple users are able to perform actions inside of the same Spark cluster in a concurrent and reliable way. This is obtained trough the use of a REST interface that can be used to call the jobs with a simple POST request.
As we can see in figure 4.2, we used Livy as a middle man between the web interface and the Spark cluster. In fact, in a production environment the Spark job will run on a cluster. To submit Spark jobs in Livy it is required firstly to start a session. In the online documentation provided on the Apache website [20] we can find a complete list of the REST API commands provided by Apache Livy. We leveraged them in order to start a session with a simple POST request, in which we specified the jar of the project and we configured some Spark options, like for example the

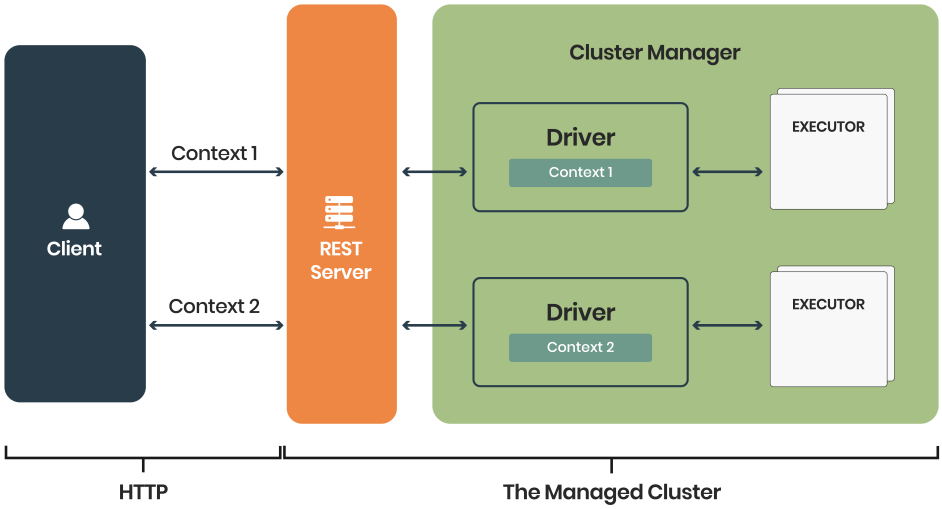

Figure 4.2: Livy architecture

number of cores dedicated to that particular session. The idea for the final project would be that each user gets a dedicated session. In this way, the administrator can choose to assign more computational power to specific users, while limiting it for others.

After the creation of the session, Livy allows us to create the so called *statements*. These are the actual commands containing the code used for launching a specific function. A session can have multiple statements. When the Spark job is completed, we can use the postman API platform in order to easily submit post requests. Then, for our local tests, in order to see the results, we only need to access the address `http://localhost:8998` in any browser to visualize all the sessions and statements submitted in postman, along with the session logs and the actual output of the job, formatted as an array of strings, as previously shown in 3.6.
We tested various anonymization rules inside of a single Livy session, querying two data sources that were deployed on Docker containers, obtaining good results. We will talk about these in more detail in chapter 5, dedicated to the final results and conclusions.

4.3 Spark Job

The Spark job was the core part of this project, on which we focused most of our efforts during the development of the PQP library. We built the project in

intelliJ, using Scala version 2.12 and sbt as a build tool. A custom sbt command was used to create the pqp deploy package, composed of a jar of the full project, configuration files, required dependencies and libraries.
Now let's get to the actual code base. We divided the project in various sub modules. Each of them has a specific purpose in our application. Let's analyze these modules one by one:

Anonymization

In this package we implemented all the methods needed to apply anonymization rules. Here we are using the ARX framework as a tool to perform privacy preserving algorithms. The main Scala trait that defines the anonymization functions is called `AnonymizationService`. This trait is extended by the `ArxAnonymizationService` class, that provides the actual implementation of the functions that were defined in the trait. Probably the most important method here is the `anonymizeDataset` function. It takes three inputs:

1. dataset: a SparkSQL DataFrame object: it contains the plain data without anonymization. In the context of the main project, this variable represents the complete result of the query that was given by the user as an input to the POST call used to submit the Livy statement.

2. metadataRequest: instance of a custom case class containing all the useful information that was passed by the user when starting the spark job. This information includes the dataset id, the scope id and the string containing the sql query that was originally submitted by the user.

3. rule: type of the rule that needs to be applied to the input dataframe before sending it back as an output.

In the anonymizeDataset function we firstly analyze the rule variable in order to understand what anonymization technique should be applied to the input dataframe (if any). After obtaining this information, we apply a custom function, according to the specific rule required. To do this we created a `ApplicationRules` trait with a function called `applyRules`, that has a specific implementation for each type of anonymization rule. In these implementation we retrieve the rule parameters from the metadata repository and we apply the rule itself. Finally, a DataFrame object is returned, containing the anonymized data as an output.

Configuration

In this module we store all the data structures needed to work with configuration data. This includes things like connection strings for the various data sources,

username and password, port number...
These information is organized in custom case classes to provide a more elegant and easy to use code. We implemented also some get functions to retrieve information easily from configuration files.

Logging

To print information during code execution we implemented a simple logger with four levels of logging importance (Debug, Info, Warn, Error).

Metadata

Here we store all the logic needed to deal with the metadata repository, like the MetadataRequest case class (to store input information like SQL query, dataset id...), the MetadataResponse (to store all the information needed by the anonymization algorithm like rule type, schema...), and other custom data structures to store other details. In addition to this, we implemented a trait containing all the custom queries to retrieve data from the metadata repository. Unit tests were implemented to make sure that all the queries were able to perform as intended, by using the embedded postgres dependency as we explained before.
Finally, since the Presto engine was not exposing the data types after performing the SQL query, we also implemented a custom function, called `extractResultSetInformation`, that was used to extract the schema of the SQL table resulting from the query, by taking as input the query string and the complete schema of the original table(obtained from the metadata repository).

Utils

Miscellaneous functions that include conversions for our custom data structures, query refinement methods, functions to extract data source properties...

Launcher

Here we store the main Spark job function. We could think about this as the main function of the PQP application. Here we have the complete end to end job written. Firstly it parses the arguments, extracting a MetadataRequest object with all the information needed (dataset id, scope id and SQL query). Then, it fetches the data by executing a custom query in the PostgreSQL metadata repository. The output is stored in a MetadataResponse object, that contains all the information needed by the anonymization algorithms (like data source information, rule type, schema with data types...).

After this process, we use the details obtained from the metadata repository to perform the query with the Presto engine. Since the output coming from Presto is formatted as a list of list of Java objects, we implemented a `createDatasetFromPrestoResult` function, capable of creating a Spark SQL DataFrame containing the data. This brings two main advantages: not only we are now capable of performing transformations of this DataFrame in the distributed Spark environment, but we are also able to store data type information inside of it.

Finally, we can send the DataFrame (that basically represents the output of the query without anonymization) to a `ArxAnonymizationService` object, that will have the goal of applying the anonymization function required by the specific rule.

4.4 Unit testing

1. **Datasource tests**: we performed test to ensure that every implemented data source was working as planned. In addition to this we tested all the metadata repository queries.
2. **Metadata tests**: In this part we tested all the queries related to the metadata repository. In addition to this we tested the extractResultSetInformation (used as we said before to parse the query and extract the presto result set schema).
3. **Anonymization tests**: here we tested all the rules, making sure that the anonymization was performed correctly on some simple dummy DataFrame objects created for the test.

4.5 Data type mapping

During the development of the PQP application we encountered some problems caused by datatype mapping. This happened because the set of data types used by Presto was not standardized in the same way as Spark did. Because of this, there was not a complete overlap between types. We solved this with an object called `DataTypesHelper` inside of our Scala code, that was used to cast incoming data to the correct type, thus avoiding implicit conversion issues.

4.6 Data sources

We initially planned to support three data sources in the final project: PostgreSQL, MongoDB and Elasticsearch. In the last iteration of code development, only

PostgreSQL and MongoDB were deployed and tested thoroughly. Query execution was efficiently performed for both of them, with similar results in terms of execution speed. We also tested them with high quantities of data, by providing a table with a million entries. The horizontal scalability of the Spark engine proved to be efficient for both of the data data sources that we analyzed.

4.7 Integration with the ARX library

As said before, we encountered some dependency problems when integrating the ARX library inside of or project. We decided to write a custom script (as shown in section B) to solve them by performing the following actions:

1. Download the ARX min package, containing only the ARX source code without any dependency attached.
2. For every element in the list of ARX dependencies, check if that dependency was already used inside of our main project.
3. If not, download it as an unmanaged dependency.

4.8 The Agile workflow

The thesis work was developed in collaboration with Agile Lab. Since 2014, the company has been creating value for its Clients by leveraging the power of Big Data and Analytics, Machine Learning, Edge AI, IoT, Low Latency Data Streaming and Domain Driven Design.The creation of this project would have not been possible without the expertise and tools provided by Agile. By working in this environment, we were able to learn a specific development workflow, that is at the core of the software development factory in AgileLab.

Gitlab is at the center of this process, allowing us to have a single source regarding the status of a project. The main values behind the software development workflow are the following:

1. Continuous improvement: The plan is not rigid, and it merges the work of everybody involved in the project implementation phase. The work of everyone must continuously converge in the development process.
2. Democracy: There is room for changes in the plan, as long as everyone involved shares the same vision.
3. Recursiveness: The process by which the work is developed and modified is the process itself.

4. Visibility: The current status of the project must always be clear, at every possible level of detail. This includes the use of:
   (a) GitLab issues
   (b) Merge requests
   (c) Code quality
5. Reproducibility: Given an output O, we want to be able to know at what stage of the project it was produced, and what was the input I that produced it.
6. Automation: The less manual procedures are needed, the better it is for the development workflow.

4.8.1 Development model

The software development is divided in phases. At the beginning, the Architect assigned to the project starts by laying out a document containing the high level design of the project (HLD). After defining the main goals of the project and the first design of a schema, the next step is to open a new GitLab repository, that will serve as a container for all development activities.

The workflow model is based on the concept of *sprint*. A project is composed of a series of sprints with the aim of creating fast release cycles by creating a due date for the end of the sprint. This allows for a faster development process, since we will focus on what has priority in the near future. The development is organized in a concentric system, based on 3 levels of granularity:

1. High level design
2. Low level design
3. Work item

This system is implemented by creating GitLab issues with the corresponding level of granularity. Issue with the same level are organized together in milestones, so we will have three types:

1. High level design milestone
2. Low level design milestone
3. Sprint milestone

An high level design milestone can contain multiple low level design milestones, that in turn are capable of containing multiple sprints inside of them

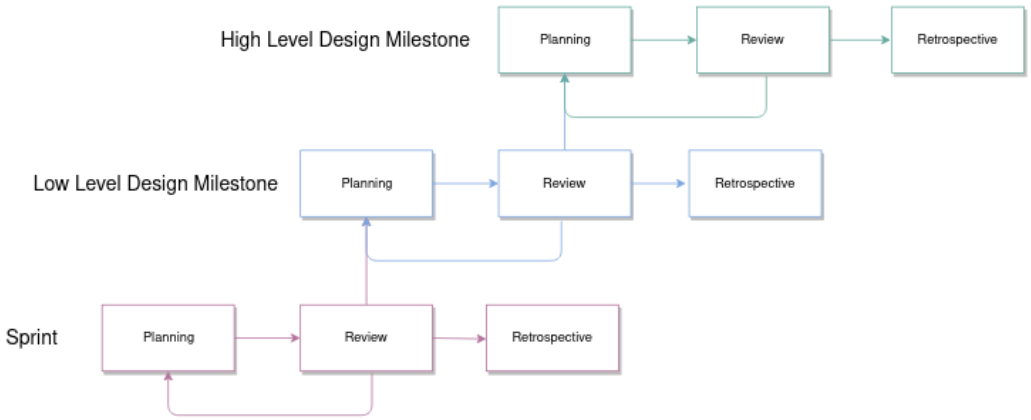

Figure 4.3: Milestone events workflow

4.8.2 Issues

It is the standard tool used by GitLab to track and manage software development. Issues should be structured as much as possible according to the INVEST model (independent, negotiable, valuable, estimable, small, testable). Let's analyze each of this 6 characteristics.

Independent

Two issues should always be autonomous between each other. If you find two correlated issues, a good idea would be to merge them together in a single one.

Negotiable

Issues should always leave room for discussion. In the description of an issue we always talk about a *proposed solution* to the problem, leaving space for opinions and suggestions on the development of a feature.

Valuable

Issues must add value for stakeholders. Making up new issues that focus on small technicalities that do not bring value are violating one of the principles of agile methodologies, which consist in providing valuable software products.

Estimable

The size of the work needed by an issue must always be somehow measurable. If this does not happen, the issue will never be planned, and it will never become part of a sprint in the future. If we cannot make an estimate due to the the lack of available information, we must iterate on the description of the issue in order to make it more clear.

Small

Issues that are too large often become impossible to work with with the right level of confidence. We would like to keep the size of an issue to be around a couple of days, and never more than a week of work. Anything bigger should be divided into smaller issues, that are easier to manage. Issues that do not fit into this definition can be defined as *Epic*, and they will last more than one sprint.

Testable

The description of the issue should provide all the details that are needed to develop tests around that feature. An issue should be considered completed only if it has been successfully tested. This is particularly true for teams using TDD (Test Driven Development). Because of this point, test coverage assumes a great importance inside of Agile projects.

4.8.3 Branching model

In order to perform version control tasks, it is important to implement a very specific git branching model, to keep a standard way of doing things when it comes to the implementation of the tasks explained in the issues.

Master branch

It is the main branch of the project. It is always updated to the latest release. In this branch we have all the code quality controls mechanisms implemented. The software obtained after compiling the master branch has still to be considered as a SNAPSHOT as it does not correspond to any formal release.

Feature branches

Feature branches are related to a particular issue, so they should be used in a short time frame. They are usually assigned to a single person and they represent an atomic unit of code review and integration within the master branch. A branch will always be correlated to a single issue. After the development of that issue

is completed and tests are working correctly, we are ready to merge the feature branch into the master. If, in the meantime, someone else already pushed new updates to the code base, we will need to rebase our branch into the main one before merging our contribution.

Release branches

When all the features required by a particular software release have been implemented and merged in the master branch, a new release should be made. To allow for the backport of new features to a version that was already released, it is also possible to integrate the feature branch with a release branch or to perform the cherry-picking of the feature branch's content inside of the release branch. The final resulting software obtained by compiling the release branch should still be viewed as a snapshot, since a release branch could be subject to backport of new features or bug fixes.

Hotfix branches

These branches are created with fast or urgent bug fixes in mind.The hotfix branches are branched from the release that needs the urgent fix, and they are integrated on there. If the fix will also be required in future releases the hotfix is cherry picked.

Tags

Tags are used to identify a certain release commit in a precise and straightforward manner. The artifact that is obtained after compiling a tag can be considered stable, and its content is well defined and traceable. The artifact version corresponds to the tag name.

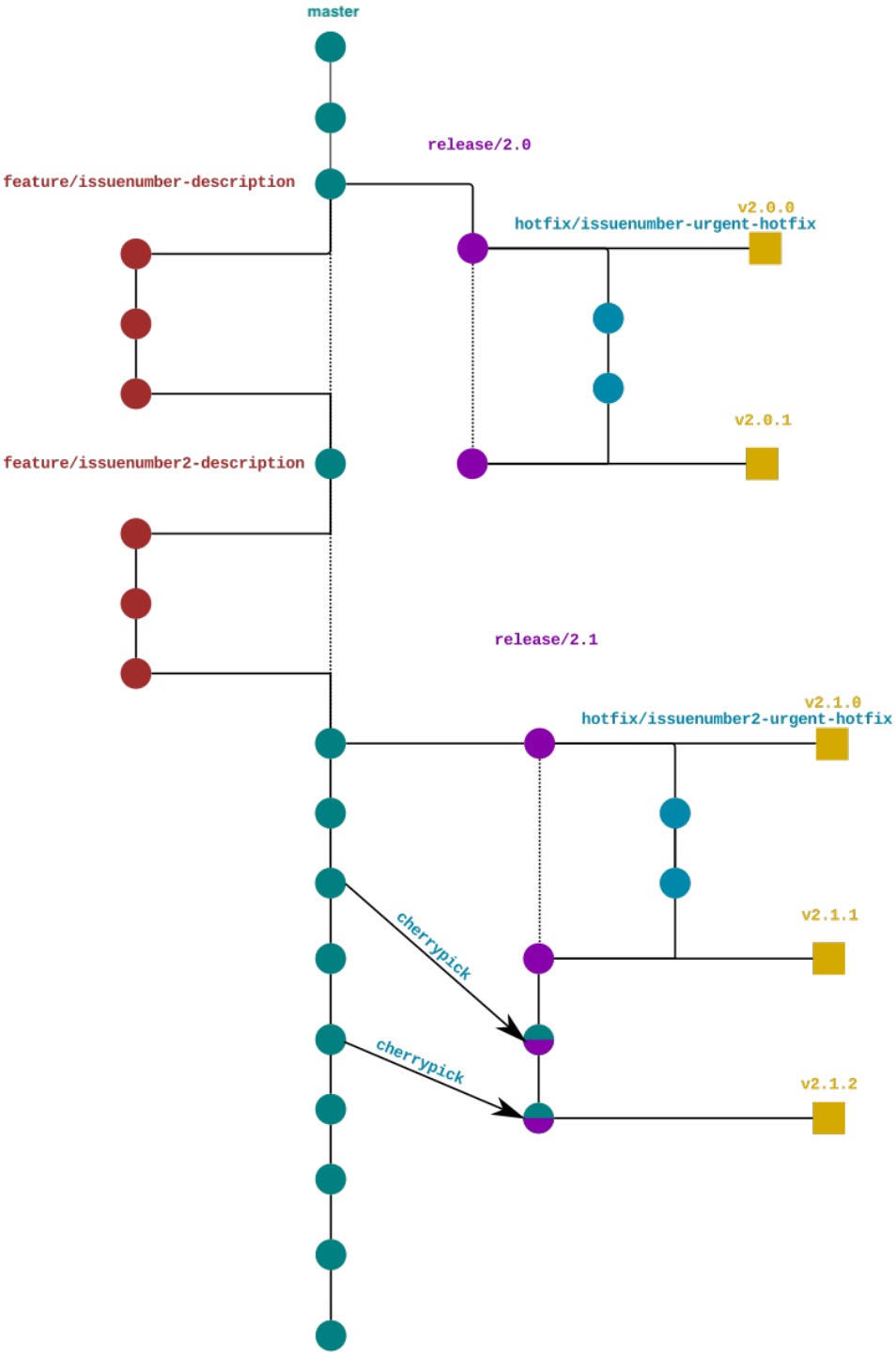

Figure 4.4: Branching model of the agile development workflow

Chapter 5

Final results

We developed a Spark job capable of querying multiple data sources, applying different anonymization techniques to the results of the query. We deployed it in a Docker environment, together with the data sources and performed some tests. We also analyzed the topic of data classification in relation to data privacy, developing a machine learning model capable of assigning labels to new data uploaded to the system. In this section we will analyze the results of the various tests that we performed on the different parts of the PQP tool.

5.1 Spark job execution results

From the Livy browser interface we are able to see details like session logs and execution time. In table 5.1 we can see the results of rule application to a sample database with a table containing data belonging to 1000 fake users. The query was always the same for all tests (`SELECT name, surname, age from users`), and thanks to the Livy web interface we were able to read the execution time for the different algorithms.
In the result table we summarized the relevant information about expected and actual result of the tests. We can see for example that, when applying group suppression with the groupMinCount variable set to 1 for the column age, we get back the original data. This is correct, since the minimum level of aggregation is always respected with 1 as the input value. However, if we set the same variable to 10, the age gets suppressed, since the algorithm is not able to find groups of at least 10 people with the same age. All the results were documented on table 5.1.

In figures 5.1, 5.2, 5.3 we can see the results on a 1000 row table of fake users that was generated automatically by using mockaroo, a web tool to create fake data for testing purposes.
As we can see from the results, all the algorithms performed as expected, with

Rule type	column	RuleParameters	Result	Time
Generalization	age	HyerarchyLevel=2	26->[0-50]	16s
Generalization	age	HyerarchyLevel=Max	26->[*]	4s
Generalization	age	HyerarchyLevel=0	26->26	5s
GroupSuppression	age	GroupMinCount=1	26->26	8s
GroupSuppression	age	GroupMinCount=10	26->*	11s
Suppression	name	Partial(Regex=.2$)	name->na**	13s
Suppression	name	Total(default val=****)	name->****	12s

Table 5.1: Spark job tests results

```
res2: Array[String] =
Array({"name":"Annora","surname":"Wallman","age":"0-50"},
{"name":"Aurea","surname":"La Vigne","age":"0-50"},
{"name":"Livy","surname":"Brandoni","age":"0-50"},
{"name":"Myrtice","surname":"Gallihaulk","age":"51-100"},
{"name":"Deana","surname":"Starie","age":"51-100"},
{"name":"Horst","surname":"Muggleton","age":"51-100"},
{"name":"Saunders","surname":"Pittway","age":"51-100"},
{"name":"Meriel","surname":"Terry","age":"51-100"},
{"name":"Rora","surname":"Scown","age":"0-50"},
{"name":"Victoria","surname":"Twizell","age":"0-50"},
{"name":"Boy","surname":"Semark","age":"0-50"},
{"name":"Andriana","surname":"McAllister","age":"51-100"},
{"name":"Katrina","surname":"Shevelin","age":"0-50"},
{"name":"Stephen","surname":"Fifoot","age":"0-50"},
{"name":"Fayina","surname"...
```

Figure 5.1: Result of generalization applied on the age column

execution times that can be considered acceptable since we were running the cluster in our local machines.

5.2 Execution on big data

All the tests described in the last section were performed on a simple data base composed of just 1000 records. We also tried to execute the queries on a table with many more entries, with the goal of assessing the performance of our tool on big quantities of data. We used a Mongo collection of 1 million documents, and we performed the same query as before, running the job both on local and YARN.

```
res1: Array[String] =
Array({"name":"Anno**","surname":"Wallman","age":17},
{"name":"Aur**","surname":"La Vigne","age":3},
{"name":"Li**","surname":"Brandoni","age":31},
{"name":"Myrti**","surname":"Gallihaulk","age":94},
{"name":"Dea**","surname":"Starie","age":53},
{"name":"Hor**","surname":"Muggleton","age":53},
{"name":"Saunde**","surname":"Pittway","age":94},
{"name":"Meri**","surname":"Terry","age":90},
{"name":"Ro**","surname":"Scown","age":10},
{"name":"Victor**","surname":"Twizell","age":37},
{"name":"B**","surname":"Semark","age":17},
{"name":"Andria**","surname":"McAllister","age":60},
{"name":"Katri**","surname":"Shevelin","age":29},
{"name":"Steph**","surname":"Fifoot","age":17},
{"name":"Fayi**","surname":"Connick","age":83},
{"name":"Ode**","surname":"Bisset","age":63}, {...
```


Figure 5.2: Partial suppression of the name column


```
res0: Array[String] =
Array({"name":"****","surname":"Wallman","age":17},
{"name":"****","surname":"La Vigne","age":3},
{"name":"****","surname":"Brandoni","age":31},
{"name":"****","surname":"Gallihaulk","age":94},
{"name":"****","surname":"Starie","age":53},
{"name":"****","surname":"Muggleton","age":53},
{"name":"****","surname":"Pittway","age":94},
{"name":"****","surname":"Terry","age":90},
{"name":"****","surname":"Scown","age":10},
{"name":"****","surname":"Twizell","age":37},
{"name":"****","surname":"Semark","age":17},
{"name":"****","surname":"McAllister","age":60},
{"name":"****","surname":"Shevelin","age":29},
{"name":"****","surname":"Fifoot","age":17},
{"name":"****","surname":"Connick","age":83},
{"name":"****","surname":"Bisset","age":63},
{"name":"****","surname":"Sande...
```


Figure 5.3: Total suppression of the name column with a custom string

Execution times were always between 20 and 30 seconds for all the queries. Spark is horizontally scalable, and while this times could be seen as a good performance for a collection of a million documents, we probably could have obtained even better results by running the pipeline on an real cluster, instead of our local machines. As a next step, we propose to execute the same job inside of an Amazon ECS cluster and to observe improvements in execution times when it comes to big quantities of data. When doing so, we will need to keep in mind intrinsic problems of Spark job execution on cluster environments, such as network latency, number of available machines, available memory...

5.3 Fingerprinting results

Some tests were performed also on the fingerprinting library, in order to assess the correct functioning of the distributed random forest classification model. The algorithm was created via the H2O Flow tool, that allowed us to perform training and validation in a simple web interface similar to the one of a Jupyter notebook.

As previously mentioned, the goal of the random forest classification tool was to make predictions on the type of data contained in the input column (names, addresses, phone numbers...), with the final goal of assigning a label to each column of an input table. The label would be directly inferred from the type of data predicted by the algorithm. For example, a phone number would be labelled as a personal identifier, a city would be labeled as a quasi identifier... and so on.
After creating the model, the Flow tool allowed us to download a Java MOJO package, that we were able to import seamlessly inside of the codebase. In figure 5.4 we can see some of the classification results obtained by feeding test columns to the distributed random forest algorithm. The test columns used were taken from real world datasets containing addresses, ages, cities, dates, names and nationalities. The random forest algorithm returns as an output a series of probabilities, one for each possible label. In this example, we printed the top 3 probabilities for each of the test columns.

```
==> addresses.res <==
Column name : addresses
______Best 3 predicted labels______
Label   Probability
address     0.5405183186111645
name        0.18022973211658566
birth Date  0.06551737195286843

==> ages.res <==
Column name : ages
______Best 3 predicted labels______
Label   Probability
age             0.9882509981459277
religion        0.003990106086540783
address 0.0033221131238941846

==> cities.res <==
Column name : cities
______Best 3 predicted labels______
Label   Probability
city    0.6120060815415082
nationality     0.1984052869423661
name    0.08514799855982469
```

```
==> dates.res <==
Column name : dates
______Best 3 predicted labels______
Label   Probability
birth Date   0.2531960264865931
gender       0.15191761589195588
day          0.1428749004446967

==> name.res <==
Column name : name
______Best 3 predicted labels______
Label   Probability
name      0.7526093633841829
city      0.12101393475103925
religion  0.05087327065428244

==> nationality.res <==
Column name : nationality
______Best 3 predicted labels______
Label   Probability
nationality     0.6632602358926135
country 0.16584772768332634
city            0.07318174465165553
```

Figure 5.4: Some results of the random forest trained on data fingerprints

Chapter 6

Conclusions

We studied the available literature regarding data anonymization techniques and open source projects to perform privacy preserving data mining tasks. We developed a software application capable of applying state of the art privacy preserving algorithms inside of a large-scale data processing environment, by using a distributed programming approach. We were able to deliver a working end to end product capable of performing queries on multiple data sources, returning an output sanitized according to some specific predefined anonymization rules.
We introduced a simple metadata repository inside of a PostgreSQL schema in order to store all the details needed by the system to apply privacy preserving tasks correctly. We provided a simple way to apply changes to said repository for users with administrator privileges. We exposed a functioning REST API, allowing end users and system administrators to interact with the Spark job from a web application.
We also explored the topic of data classification, with the goal of predicting the level of sensitivity for newly added tables, thanks to a distributed random forest algorithm.

For the future development of the project we plan on deploying the entire project on a cloud environment, with the help of Kubernetes to handle the containers. This should be mainly an integration problem, since the Spark job was written from the beginning with the goal of working both on premise and on cloud. We also plan to integrate ElasticSearch as a third data source inside of a Docker container. Other future improvements to the project include the addition of new privacy models to the set of available anonymization rules. For example we could implement some of the techniques that we analyzed in the previous chapters, such as l-diversity, t-closeness and so on.

Other future development plans include working on the integration between the

fingerprinting job and the PQP library, running the classification algorithm at fixed intervals during the day or, as an alternative, every time a new table gets uploaded to the system. The data privacy administrator will then be able to see the classification and make changes if needed. I plan on continuing the development of these new features in the next months, as I will keep on working in Agile Lab after the end of my academic career.

Appendix A

Scala code snippets

```
1     case class SuppressionData(
2     suppressionType: SuppressionType, // Total or Partial
3     columns: Array[String],           // list of columns to suppress
4     defaultValue: String,             // string to use when
    suppressing (for example "*")
5     partialRegex: Option[String],     // In case of partial
    suppression, to select what to suppress
6     )
```

```
1     case class GeneralizationData(
2     name: String, // name of the column
3     intervalBased: Boolean, // true if it is a numerical hierarchy
4     hierarchyLevel: Int, // level at which generalization is
    evaluated
5     hierarchy: Map[String, String] // generalizaiton hierarchy
6     )
```

Appendix B

Arx setup

```
#!/usr/bin/env bash
set -o errexit
set -o nounset
set -o xtrace

TMP="tmp"
dependencies=( "libarx-3.9.0-min.jar|https://github.com/arx-deidentifier/arx/releases/download/v3.9.0/libarx-3.9.0-min.jar"
"jhpl-0.0.1.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/jhpl/jhpl-0.0.1.jar"
"newtonraphson-0.0.1.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/newtonraphson/newtonraphson-0.0.1.jar"
"objectselector-0.1-lib.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/objectselector/objectselector-0.1-lib.jar"
"mahout-core-0.9.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/mahout/mahout-core-0.9.jar"
"mahout-math-0.11.1.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/mahout/mahout-math-0.11.1.jar"
"colt-1.2.0.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/colt/colt-1.2.0.jar"
"hppc-0.6.0.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/hppc/hppc-0.6.0.jar"
"commons-math3-3.6.1.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/commons/commons-math3-3.6.1.jar"
"commons-validator-1.4.1.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/commons/commons-validator-1.4.1.jar"
"exp4j-0.4.8.jar|https://github.com/arx-deidentifier/arx/raw/master/lib/ant/exp4j/exp4j-0.4.8.jar"
"smile-core-1.4.0.jar|https://search.maven.org/remotecontent?filepath=com/github/haifengl/smile-core/1.4.0/smile-core-1.4.0.jar" )

```

```
[[ -d $TMP ]] && rm -r $TMP
mkdir $TMP

echo "check environment..."
if [[ $OSTYPE == 'darwin'* ]]; then
       echo "macOS found..."
       #brew install zip
    else
       echo "*nix environment found..."
       #apt install zip
fi
echo "check existing of lib folder..."
if [[ ! -d "./lib" ]]; then
    echo "lib folder does not exist, creating it... "
    mkdir -p ./lib
fi
echo "check dependencies...."
cd $TMP
for kv in "${dependencies[@]}"; do
  dependency=${kv%%|*}
  url=${kv##*|}
  if [ -f "../lib/$dependency" ]; then
    echo "$dependency found!"
  else
    echo "Downloading $dependency ..."
    curl -vLJO -H 'Accept: application/octet-stream' ${url}
    mv $dependency ../lib;
  fi;
done
cd ..
rm -rf ./$TMP
echo "Libarx installed successfully"
```

Bibliography

[1] Latanya Sweeney. «Weaving Technology and Policy Together to Maintain Confidentiality». In: *The Journal of Law, Medicine & Ethics* 25.2-3 (1997). PMID: 11066504, pp. 98–110. URL: https://doi.org/10.1111/j.1748-720X.1997.tb01885.x (cit. on p. 1).

[2] Gina Stevens. «Data Security Breach Notification Laws». In: (Apr. 2012), p. 6 (cit. on p. 3).

[3] Pierangela Samarati and Latanya Sweeney. «Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression». In: (1998) (cit. on pp. 4, 9).

[4] Marmar Orooji and Gerald M. Knapp. *Improving Suppression to Reduce Disclosure Risk and Enhance Data Utility*. 2019. arXiv: 1901.00716 [cs.DB] (cit. on p. 6).

[5] Vanessa Ayala-Rivera, Patrick Mcdonagh, Thomas Cerqueus, and Liam Murphy. «A Systematic Comparison and Evaluation of k-Anonymization Algorithms for Practitioners». In: *Transactions on Data Privacy* 7 (Dec. 2014), pp. 337–370 (cit. on p. 7).

[6] K. LeFevre, D.J. DeWitt, and R. Ramakrishnan. «Mondrian Multidimensional K-Anonymity». In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, pp. 25–25. DOI: 10.1109/ICDE.2006.101 (cit. on p. 10).

[7] Qiyuan Gong. *Mondrian*. https://github.com/qiyuangong/Mondrian. 2015 (cit. on p. 11).

[8] R.J. Bayardo and Rakesh Agrawal. «Data privacy through optimal k anonymization». In: *21st International Conference on Data Engineering (ICDE'05)*. 2005, pp. 217–228. DOI: 10.1109/ICDE.2005.42 (cit. on p. 12).

[9] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. «L-diversity: Privacy beyond k-anonymity». English (US). In: *ACM Transactions on Knowledge Discovery from Data* 1.1 (Mar. 2007). ISSN: 1556-4681. DOI: 10.1145/1217299.1217302 (cit. on pp. 12, 14).

[10] Dr Kumar. «t-Closeness: Privacy Beyond k-Anonymity and-Diversity». In: Jan. 2011 (cit. on p. 15).

[11] Justin Brickell and Vitaly Shmatikov. «The cost of privacy: destruction of data-mining utility in anonymized data publishing». In: *KDD*. 2008 (cit. on p. 16).

[12] Jianneng Cao and Panagiotis Karras. «Publishing Microdata with a Robust Privacy Guarantee». In: *CoRR* abs/1208.0220 (2012). arXiv: 1208.0220. URL: http://arxiv.org/abs/1208.0220 (cit. on p. 16).

[13] Mehmet Nergiz, Maurizio Atzori, and Chris Clifton. «Hiding the presence of individuals from shared databases». In: Jan. 2007, pp. 665–676. DOI: 10.1145/1247480.1247554 (cit. on p. 16).

[14] Fabian Prasser and Florian Kohlmayer. «Putting Statistical Disclosure Control into Practice: The ARX Data Anonymization Tool». In: *Medical Data Privacy Handbook*. Ed. by Aris Gkoulalas-Divanis and Grigorios Loukides. Cham: Springer International Publishing, 2015, pp. 111–148. ISBN: 978-3-319-23633-9. DOI: 10.1007/978-3-319-23633-9_6. URL: https://doi.org/10.1007/978-3-319-23633-9_6 (cit. on p. 18).

[15] *General Data Protection Regulation*. European Commission. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679 (cit. on p. 22).

[16] Prasser. *arx-deidentifier*. https://github.com/arx-deidentifier/arx. 2014 (cit. on pp. 40, 42).

[17] Carlos Becker. *Dealing with Maven dependency hell*. 2016. URL: https://carlosbecker.com/posts/maven-dependency-hell/ (cit. on p. 43).

[18] Jendrik Johannes. *Why libraries like Guava need more than POMs*. 2020. URL: https://blog.gradle.org/guava (cit. on p. 43).

[19] Zonky. *embedded-postgres*. https://github.com/zonkyio/embedded-postgres. 2012 (cit. on p. 44).

[20] The Apache software foundation. *Livy docs - REST API*. 2021. URL: https://livy.incubator.apache.org/docs/latest/rest-api.html (cit. on p. 46).