Anonymization of production data to create functioning and useful test environment data

Niki Norrman
Master's thesis in Computer Science
Supervisor: Marina Waldén
Faculty of Science and Engineering
Åbo Akademi University
2023

Abstract

This thesis will investigate different methods to anonymize production data to inject it or parts of it into the test environment. The different methods will be analyzed from different perspectives and some of them will be implemented in the implementation stage. Each method will be analyzed on a general plane but also from the point of view of a specific use case. The different methods might have to be used simultaneously for different parts of the process where different legal or technical requirements exist. Some methods (such as training an AI model) are more time-consuming to implement than others and, thus, they were not considered worth implementing in this thesis. We will also go through which methods work for our specific use case and which support our goals the most. Those will not be implemented but they will be considered in the comparison section of this thesis. Finally, this thesis will compare and summarize the analyses of the different methods in the summary section, and use cases will be suggested for the specific method for future use.

Keywords: IT, information technology, Anonymization, GDPR, Relational, Data, Testing

Preface

Abbreviations and Terms

- General Data Protection Regulation (GDPR)
- Health Insurance Portability and Accountability Act (HIPAA)
- America Online (AOL)
- Graphical User Interface (GUI)
- Command Line Interface (CLI)
- Machine Learning (ML)
- Generative Adversarial Network (GAN)
- Anonymization through Data Synthesis using a GAN (ADS-GAN)
- Minimum Viable Product (MVP)
- Primary Key (PK)
- Foreign Key (FK)
- Amazon Web Services (AWS)
- Structured Query Language (SQL)

November 17, 2023

Table of Contents

1 Introduction **1**

2 GDPR **3**

3 Perspectives of Anonymization **5**

3.1 Risks 5

3.2 K-anonymity 6

3.3 Anonymization in Social Networks 7

3.3.1 Utility vs identifiability 10

4 Anonymization Methods **12**

4.1 µ-Argus 12

4.2 Token-based hashing anonymization 14

4.3 Generative Adversarial Network solution 15

4.3.1 Method complexity 17

4.3.2 Relational database compatability 18

5 Anonymization Project Requirements **19**

6 Comparison **21**

6.1 µ-Argus 21

6.2 Token-based hashing anonymization 21

6.3 Generative Adversarial Network solution 23

7 Implementation **24**

7.0.1 Implementation steps 24

7.1 Analysis 28

7.1.1 Goals and priorities 28

7.1.2 Binning 29

7.1.3 Noise for numbers and dates 31

7.2 Triggering and fetching data 32

7.3 Applying methods . . . 33
7.3.1 Basic binning method . . . 33
7.3.2 Binning accuracy . . . 35
7.3.3 Testing speed and accuracy . . . 36
7.3.4 Improved accuracy and speed with seedrandom . . . 38
7.3.5 Shortening strings . . . 39
7.3.6 Using the binning method for non-ID values . . . 43
7.3.7 Random noise for date values . . . 44
7.4 Improving the code . . . 46
7.4.1 Testing and code quality . . . 46
7.4.2 Execution time at scale . . . 48
7.4.3 Faster performance without seeds . . . 50

8 Discussion 52
8.1 Execution time related issues . . . 52
8.1.1 Faker-js . . . 53
8.2 Database and connection related issues . . . 53
8.3 Test structure for the anonymizer . . . 54
8.4 To seed or not to seed? . . . 54

9 Conclusion 57

10 Anonymisering av produktionsdata för att skapa fungerande och användbar testmiljödata 58
10.1 Introduktion . . . 58
10.2 Laglighet . . . 58
10.3 Perspektiv på anonymisering . . . 59
10.4 Anonymiseringsmetoder . . . 60
10.5 Projektkrav . . . 61
10.6 Jämförelse av metoder . . . 61
10.7 Implementering . . . 62

10.8 Diskussion 63
10.9 Slutsats 64

1 Introduction

The reason for anonymization of many of the sources [1, 2, 3] has been to make production data safe for release to the public with focus on it being useful for various research purposes. However, the main goal for the implementation in this thesis work is to create better test data from the production data that on some level represents what exists in production, while people and companies are not recognizable, i.e. complying with GDPR. The implementation will be used to release anonymized production data most likely for internal use, meaning advanced methods of analyzing the anonymized data to correlate to real-world data might not be too relevant. For the purpose of the literary work, though, it can be interesting to study how to avoid those more advanced methods, even if they might not be necessary for our specific implementation needs.

The main reason for the need to generate test data, and to find a good method to do it, is to have proper data in a non-production environment. There can be many benefits in having a representative dataset in a non-production environment. Test environments are often not exact copies of production environments. Reviewing data tables and loading times shows that some tables usually are less populated in test, while others can have more data than production. When developing systems such as search queries the developer usually does not have the ability to test their new developed features in production before the feature has been released by definition. This can often lead to developers not giving the correct attention to possibly badly optimized search queries that cause the end users to spend unexpected amounts of time waiting for results, or even having queries timing out. These problems are often easier to iron out if caught early, since data tables are often more difficult to change once in production.

Another benefit of generating anonymized data for test environments from real production data can include the quality or realness of the dataset. A common occurrence in many test datasets are properties such as “Test property 1” and “Bob’s Test property, do not touch!”. These properties are often created to test a specific

feature by a specific tester or developer. Sometimes these properties are incorrectly linked to other parts of the complete dataset or are incomplete in other ways. These test properties are unrepresentative of real data in production.

If the data in the test environments is of low enough quality, usage of the production environment for testing or development purposes might occur. This brings numerous problems. The developer or tester could unintentionally affect customer data and cause problems. The developer or tester will in this case be able to see real-world user data unnecessarily, which might have implications from a GDPR or ethical perspective. Should a developer be able to access data meant for the customer? Probably not, at least not unnecessarily. Having unnecessary access to production environments also increases the possibility of a data leak. For these purposes, it is important for us to be able to find a good method to generate anonymized data from production data for use in a test environment.

2 GDPR

The General Data Protection Regulation (GDPR) is a piece of regulation adopted by the European Union (EU) and is being adopted in the United States of America (USA)[4]. Although GDPR is an EU law, it is said to have a global impact since any company that targets the European market must follow it[5, 6].

The GDPR was introduced in 2017 in the EU, and was enforced across all EU Member States from the 25th of May 2018. The GDPR was created from a philosophical approach to protection of people's data. The EU, having human rights as an important principle, has decided that data privacy is an important part of human rights as it relates to privacy [6]. Due to this, the GDPR was introduced. [6]

In our specific case, we are interested in the GDPR, since we are interested in the European market. The reason we must be aware of the GDPR is to know where the legal boundaries are drawn. This will allow us to know what red lines have been drawn, so we know what the minimum level of anonymization will have to fulfill.

The GDPR gives its users the right to withdraw their consent from companies to hold their data and the right to demand that the companies delete all the data about the user. The regulation means that data handlers and controllers must be designed from the start to protect people's personal data. Failure to follow these GDPR rules can cause substantial fines for the company in violation. The GDPR has implications mainly for actors in Europe, but even companies that do not have a presence in Europe, or do not handle European user data, should consider the GDPR in their IT solutions, if they have global aspirations. [5]

According to J. Yoon *et al.* [4], data protection regulations such as the GDPR and the Health Insurance Portability and Accountability Act (HIPAA) do not provide clear guidance on anonymization of personally identifiable data. Article 4 in the GDPR defines personal data as the following:

"any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an 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" [7, 4]

He Li, Lu Yu and Wu He [5] wrote that personal data is defined as any data that can be used to identify an individual. This includes data such as names, email addresses, social security numbers, IP addresses, telephone numbers, location data, birth dates and others, such as those mentioned in the paragraph above. This data is protected for every EU resident, even if they do not reside in the EU [6].

3 Perspectives of Anonymization

In this chapter, we will review two short examples from the early 2000's when people's personal data was badly anonymized and given to the public. These examples will be used to explain a few concepts regarding anonymization. These discussions will build a basis on why anonymization is important from a non-legal perspective, and methods of how anonymization can be measured.

3.1 Risks

In multiple cases in the early 2000's [8], big companies have attempted to release actual user data for research purposes. The companies have thought that releasing their user data after implementing simple anonymization of only identifying data, such as the usernames, would be enough to avoid controversy. In the cases of America Online (AOL) and Netflix in 2006 releasing their user data for research purposes, The *New York Times* revealed that users could be identified even when their usernames were anonymized. The AOL data contained user queries and their timestamps while the Netflix contained users' movie ratings and their timestamps.

In the case of the AOL dataset release, AOL ended up apologizing for the release only nine days after the dataset was made public [2]. A user's search queries were used to identify family members whom the user had queried for, in addition to local business searches. This gave the *New York Times* the home county and the family name of the user from where the user was identified due to the name not being too common in the county.

In the case of the Netflix dataset merely looking at ratings of less popularly rated movies "...it was shown that 84% of the subscribers could be uniquely identified..." [8]. In [9], it was elaborated that if the attacker knows a little bit about a user whose data was anonymized in the dataset, the attacker can identify the user from the anonymized data and learn about the person's personal sensitive data.

Risks associated with this kind of breach of anonymity can be numerous. In the case of AOL's search engine query logs, they can reveal much about the state

of mind or sensitive personal information of the user, such as locations, names of companies, inappropriate searches, and revealing sensitive information. The revealed search queries can cover topics ranging from pornographic material to employment, murder, and even suicide. This data can be a breach of privacy if released, especially if the data can be traced to individual persons. Even when some information gained from anonymous data, or queries not associated with a session-id can be considered sensitive, these are not always "safe" even to that level. Related queries can be tracked from similarity of search topics, while some search queries might contain de-anonymizable or non-anonymized search terms which can connect the queries to individuals. [2]

3.2 K-anonymity

The reason these datasets failed was that the data was not *k-anonymous*. A dataset being k-anonymous means that every user's quasi-identifying dataset is identical to the quasi-identifying dataset of at least k-1 other users' quasi-identifying datasets. The idea with k-anonymity is that when every user has at least k identical sets, the user's data will be private[8]. The benefits of k-anonymity makes the anonymized data such that it is not possible to trace any particular piece of data to one specific individual, but to k other possible matches. For a big enough k, there will be a level of privacy provided[2]. There can be many ways to achieve k-anonymity, but if we want truly representative data, we cannot merely remove every item from each user, or create all the possible combinations to drown out the user's data.

Interesting about the *New York Times* research was that they did not use the timestamps associated with the data to resolve which user what data was associated with. Having exact timestamps in anonymized data is not very good when considering k-anonymity, since that increases the number of possible combinations considerably, especially since the timestamps are associated with the ratings or queries in our two examples. If we include the timestamps in our data, k-anonymity might be easier to reach if we, for example, round off the timestamps to the closest day or month. A similar method is used in programs such as µ-Argus, for example [1].

As in that example, anonymization algorithms usually achieve k-anonymization by generalizing and suppressing quasi-identifying values in the data. As mentioned above, an extreme way to do this is by making all the data identical, or by suppressing them, however, in most cases, that is not particularly useful. For an algorithm to preserve maximal utility, while also attempting to achieve some level of k-anonymization, the algorithm must do it by performing the minimal number of generalizations and suppressions. However, utility is subjective and depending on what the data is meant to be used for. A limited number of data points when building a machine learning model might lead to underfitting of the machine learning model, for example [10]. Underfitting in data science means that the algorithm does not successfully model the effect of the inputs on the outputs.

An interesting thought about the AOL case is that in [11] it is suggested that free text can be anonymized by creating regular expressions from identifying parameters. If this could have been done to, for example the names of the relatives of the user, their data might have been at least somewhat more secure.

3.3 Anonymization in Social Networks

Netflix and AOL are not the only big companies that have released, or have the need to release, their user data for the public. Big social network companies possess vast quantities of user data (See Figure 1) that they wish to utilize in one way or another. In addition to the whims of these social network companies, the data is of interest to the people the data is composed of, and to third parties such as marketing, advertising, data collection, further resale or even malicious intents. [12]

Social network companies have vast amounts of data about their users and services. They can use the data to create different kinds of features for their services or give it to third parties. If the company decides to give their data to a third party, they might want to consider the privacy of their users. In a previous section (3.2) of this chapter, we discussed that simple anonymization steps, such as removing directly identifiable information from user data, are not sufficient to prevent the risk of re-identification. [12]

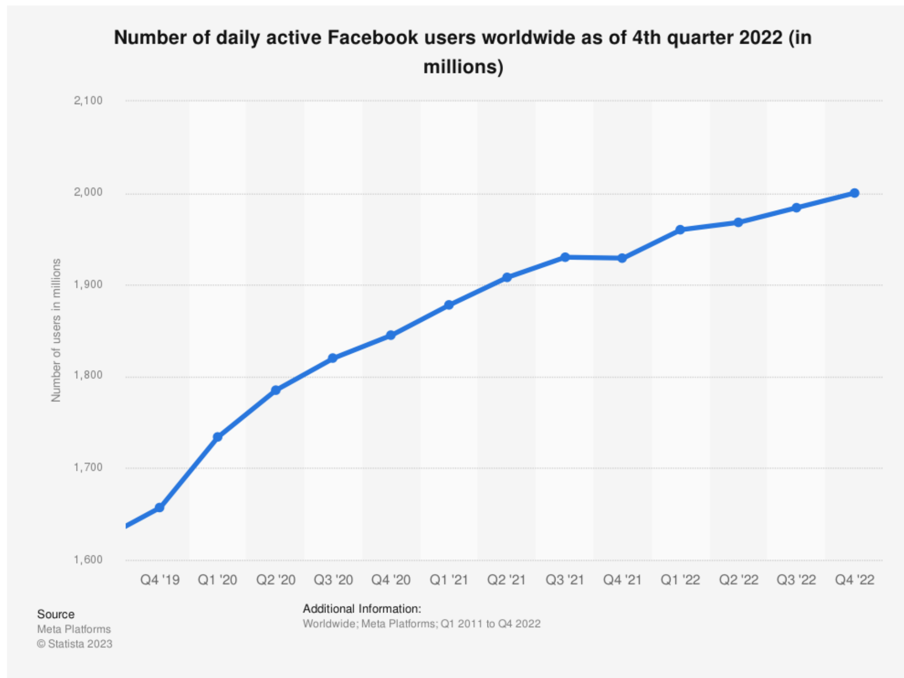

Figure 1: Facebook alone, in Q3 of 2022, had two billion active daily users and has grown by about 20% since the beginning of the pandemic. Monthly active users are reaching three billion. [13]

There can be many benefits in releasing some type of anonymized version of social network data for research purposes or even marketing. If the data is released non-anonymized, these companies will create a severe privacy threat to their users. However, if the company does not release the data, researchers will not be able to analyze the data. While many actors might have their reasons to release user data, users of these social networks might be interested in not being identified in any possible published data. Malicious actors can use published user data to identify sensitive information about users who might want to keep their data private. A graph structure that can be generated from this type of data can introduce new ways to threaten user privacy. [12]

These companies have data that contain sensitive information. The data can include anything from user interests, friends, groups, relations, or even financial exchanges, and this data can be used to build a network graph of nodes and edges. This type of graph is called a sociogram. The nodes represent actors such as the users of the social network, and the edges represent their relations such as who is

friends with whom. Edges can have different kinds of labels, such as "friendships", "partners" or "colleague" and they can be directional, for example, to indicate who is whose parent or child, and they can have weights. [12]

Sociograms can also be represented in matrix form. Matrices can be used as mathematical and computational tools to process the data to find patterns or to summarize it. One form of matrix representation of a social network is the adjacency matrix. Such a matrix can be used to represent valued relations. The complexity for a matrix to describe a social network of size n is n x n creating a matrix where the relations of the i-th and j-th nodes are represented by the cells i, j and j, i. Because of the structure of the matrix, lookup operations can be performed quickly, making adjacency matrices an efficient way to analyze relationships in large datasets. With signed and valued relations between nodes, matrices are not the most appropriate way to represent the networks, as that would require one matrix for each type of relationship. [12]

Sociograms themselves can be used to represent more complex data. The graph will consist of the nodes and their edges. Each node can have additional attributes, such as name, telephone number and age, while the edges can be labeled so as to denote different types of relations, such as the familiar relations of the nodes, or to describe a friendship. [12]

To avoid problems with bad anonymization we must first understand the actual data. According to Ouafae *et al.* [12], social network data cannot simply be anonymized by applying k-anonymity. In relational databases, the privacy attacks come from the quasi-identifiers being used to recognize individuals. In social networks, these attacks can come from for example neighbourhood graphs which can be used to identify individuals. In relational graphs, tuples can be anonymized without the rest of the data being affected, while in social networks, adding data to the graph will also affect the neighbourhoods of other nodes in the neighbourhood. [12] This can be an interesting factor to consider when implementing anonymization of our data.

3.3.1 Utility vs identifiability

Again, as mentioned in section 3.2, anonymization reduces the utility of the resulting data [12], so we would want to consider what level of anonymization is best for our specific use case. In this case, we might not be interested in perfectly representing the social networks between our data points, if our resulting anonymized data will not be used to study such relations. We can consider and weigh how important it is to have this data, thinking of the utility our use case will have versus the possible risk of re-identification it will introduce.

Perhaps, as a generic example, if we were to for example sell our data to a research company interested in studying possible relational patterns in social network data, we would have to find a way to actually protect users' anonymity in some way, while also preserving the utility of the relational social network of the data. Conversely, if we are using the resulting anonymized data to, for example, have mock data to test if new features in a new release for the system works, certain relations or network aspects in our data might not be of interest for that use case.

For different elements in our data, there can be different levels of identifiability. On the one hand, an element can be of high utility for a specific use case and cannot be anonymized at all and must thus be present in its unaltered form, while on the other hand, the piece of data can be of such low importance or utility that it can entirely be removed without negatively affecting the utility in our use case. [12]

Another viewpoint regarding anonymization risk and usefulness is if the recipient of the data knows the level of anonymization used. Knowing it, the recipient can better understand it, but it can also introduce clues towards the risk of re-identification [12]. Ouafae *et al.* [12] suggest that there are two types of utility in social networks. The first type of utility comes from overarching characteristics of the graph, such as the spread of the number of connections each individual has within a section of the social network, or the extent to which individuals form tightly-knit groups, as measured by clustering coefficients within the entire network. The other type

is aggregate network queries, where an aggregate of paths fulfilling a conditional is queried from the data. An example of this could be computing the average distance in relations of two types of nodes, i.e. how many steps in the sociograph from a programmer there are on average to a lawyer.

Ouafae *et al.* [12] write that social network anonymization is a complex problem, since they are much more complex than simple relational datasets. When anonymizing simple relational datasets we can study established models of anonymization such as k-anonymity to measure the level of privacy. In social networks, though, we have more vectors of attack than for example quasi-identifying information. As we established earlier, social networks have neighbourhood graphs, which means if one tuple of data is changed, the change will be seen in the rest of the neighbourhood graphs of the neighbouring nodes.

4 Anonymization Methods

Many different methods exist to anonymize data, but many of them are designed for specific use cases or specific organizations, such as healthcare datasets [1, 4]. This means that we should study a number of anonymization methods, and based on what we learn, we can use parts of these methods in the implementation chapter of this thesis to generate anonymized production data for test usage.

This section will evaluate various anonymization methods to distill insights relevant to our use case. Each method will be delineated with a succinct evaluation of its strengths and weaknesses. This will enable the reader to rapidly acquire a general understanding of the potential benefits and limitations associated with each respective method.

4.1 µ-Argus

The Windows program µ-Argus is used to anonymize data created by the European Computational Aspects of Statistical Confidentiality between the years 2000 and 2003 and has been further developed since then [1]. The name µ-Argus comes from the Greek myth of Argus, a giant with thousands of eyes transformed into a peacock making it more difficult to discover. This is probably a reference to k-anonymity. The program was developed to make it easier for institutions to modify data to create releasable "safe enough" data with minimal data loss for use by the public and researchers. [3]

The interactive application µ-Argus is used by different state and national statistical institutes globally [1, 3]. Many users of µ-Argus have automated the process while others have not. [1]

The µ-Argus tool was made to anonymize microdata files. Microdata is unanonymized data from surveys, censuses, and administrative systems and is used for official statistics to produce aggregated information often in a relational tabular format with

each row representing individual entries. In some contexts, microdata is protected due to its sensitive nature. [12]

Usage of the tool consists of the following steps. The user will import their microdata file into µ-Argus. After this, the user must define metadata relating to the file before continuing. The user will then define indirect identifiers and sensitive variables. A minimum risk acceptance must first be selected for these values. Then, the possibility of data re-identification using probabilistic methods can be studied using µ-Argus. Finally, the risk of re-identification can be reduced with several different methods employed by µ-Argus. Anonymization choices include the addition of noise, micro-aggregation (grouping data into small aggregates to protect individual values), post-randomization (applying random noise to data after collection), rounding, and data swapping. After these methods are applied, a residual disclosure risk estimation is conducted, after which a safe, or anonymized version of the original microdata file, can be exported. [1]

The study conducted by Bergeat *et al.* [1] regarding the anonymization of French medical record data highlights certain limitations associated with µ-Argus. The findings suggest that µ-Argus does not conform to the standards outlined by the HIPAA and lacks regular updates. These factors potentially compromise the effectiveness of anonymization, depending on the desired level of anonymization. However, it is important to note that if the context involves non-medical data, data outside the United States, or there are no plans to publicly release the data, these limitations may be inconsequential for that specific use case.

The work on µ-Argus has for several years been continued by de Wolf in an open-source manner [1, 14]. Constant development has taken place on the GitHub page [14] with the latest version tag being created on 2022-07-26.

An advantage of µ-Argus, among other factors, is that it is an existing program. No coding work needs to be done to apply µ-Argus to a microdata file. This might not be optimal for all use cases, though. µ-Argus is easy to feed data to. For an average user who is familiar with Windows applications, a Graphical User Interface (GUI) can be easier to learn compared to a command line interface (CLI) program. µ-Argus

has many anonymization features that it can implement in the user's microdata files.

Drawbacks of using µ-Argus include it being originally designed for use on microdata. While microdata is a common format for many statistical institutes, it is not used by everyone [12]. For a use case where the data is not in a microdata format, a translation step must be introduced to make use of µ-Argus. The program is not HIPAA compliant [1]. For use cases where HIPAA is relevant, µ-Argus will not be the tool to choose. Note that this might have changed since Bergeat *et al.* [1] wrote their paper, as de Wolf [14] has delivered updated versions for eight years at this point.

4.2 Token-based hashing anonymization

Kumar *et al.* [2] made a study on token-based hashing methods from the perspective of the dangers of using them. In their study, they suggest that k-anonymity-based methods are better. However, this study does make some assumptions about the attacker having a level of statistical knowledge of the targeted data.

The data the study [2] used to reverse-engineer hashes was in the style of the AOL dataset, i.e., search query logs. The article suggests that if as an attacker one has access to a non-hashed query, it is trivial to invert the hashing.

The data in this system were hashed by each query log having its own hashing token. This token is then used to hash identifying parameters in the logs to make them anonymized.

The method used by Kumar *et al.* [2] requires the attacker to have a significant amount of domain knowledge. It also requires the use of a reference dataset of similar caliber, with data that has a sufficiently large rate of co-occurrences. In the context of query logs, the logs would have to contain words or numbers that often occur together, such as “may” and “april”, or “heel” and “toe”, as Kumar *et al.* [2] found in their data.

These known patterns, such as common names and co-occurrences, can be used in an algorithm to find the token and, thus, de-anonymize the hashed data. Kumar *et al.* [2] thus propose that data anonymized in this way is not a perfect solution to

protect user data. This means that this method should not be used as a safe method to release anonymized data to the public.

An advantage of hashing is that it is relatively easy to implement. Applying a hashing algorithm to a string is a relatively simple task. It would also result in deterministic results [15], which would be useful in a case where we want to have a somewhat static test environment.

A drawback of using hashing is that it might not be safe. The anonymized data can be reverse-engineered, if an attacker has statistical knowledge of the data. Human input is also required. Choosing which fields are to be anonymized in this way might lead to mistakes.

4.3 Generative Adversarial Network solution

As machine learning has become more popular recently, we can find sources [4, 10, 16] also for implementations of machine learning to generate synthetic anonymized data [10].

Using a machine learning (ML) method can be a good way to generate anonymous synthetic data from a highly complex original dataset [4]. The synthetic nature of the generated data can mean loss of rare occurrences, e.g. rare medical conditions can be difficult to replicate in patient data without exactly replicating that patient's data [4]. Synthetic data generation results in data in which it will be harder to re-identify the original subjects who contributed their personal data [16].

Yoon *et al.* [4] created a modified Generative Adversarial Network (GAN) that they call anonymization through data synthesis using a generative adversarial network, or ADS-GAN. As the name suggests, a GAN has two parts, a generative and an adversarial part. These two parts face off against each other to refine the generative network to create output that looks like the original data but is not exactly similar. Commonly, GANs are used, for example, in image generation. In ADS-GAN, the network receives an additional dynamic input in the form of values of a set of conditioning variables. This improves the data quality, as well as makes it more unique. In essence, this means that the discriminator knows what the original data is for each

generated synthetic dataset. This makes the output of ADS-GAN quite resistant to a possible identity attack, such as those used in the AOL and Netflix cases.

In more traditional methods for anonymization, a requirement to start the process might be to determine the identifiability of specific properties in the dataset. One way of classifying identifiability includes four different categories, which are (1) directly identifying such elements as the name or social security number of a user, (2) indirectly identifying such elements as date of birth or phone number. Yoon *et al.* [4] suggest the classes of (3) potentially identifiable information such as dates of visit (to a hospital for example) and rare traits (such as a rare condition), and (4) non-identifiable information including test results and specific codes. This task can increase in difficulty depending on the complexity of the original dataset to be anonymized. In theory, specific fields can be misidentified in terms of their identifiability class. This means that due to implementation errors, certain data may pose challenges in maintaining anonymity. In addition to possible user errors, methods that are used to achieve this can be highly subjective or hardly defined. Benefits of using ML techniques to create synthetic anonymized data from complex datasets include the fact that this task can be done by a "simple" mathematical method, removing the human, assuming the mathematical method is sound. [4]

The two components of ADS-GAN optimize themselves against each other to generate results that are at the same time similar, but still different enough from the original material. The person running the ADS-GAN system has control over what to set the desired identifiability to, having to choose between reduced identifiability and increased quality. This identifiability is not a user-controlled parameter in other GANs that Yoon *et al.* [4] compared ADS-GAN with to benchmark the quality of ADS-GAN. Yoon *et al.* [4] benchmarked ADS-GAN against four other GANs with a range of identifiability settings. In the results reducing the identifiability value in all GANs gives results that are more dissimilar to the original data. In all cases, ADS-GAN demonstrated better performance, having the lowest distance scores from the original data compared to the other GANs. This benchmark was run on four different available medical datasets.

Yoon *et al.* [4] conclude their paper by suggesting that the ADS-GAN worked and they propose ADS-GAN can be used as a safe, legal, and ethical method for publishing data. They believe that there is a degree of trust that the data is not identifiable, since they used mathematical formulas to define identifiability. This means that it would not be possible to identify the original data from the anonymized set, but the resulting synthetic data should still be good enough for use. With an original dataset size of 5000, Yoon *et al.* [4] claim that ADS-GAN would produce good data. They also claim that ADS-GAN fulfils the requirements of k-Anonymity by defining similarity as the corresponding definition of indistinguishable. They claim it satisfies k-anonymity with a k value of 2 due to the identifiability parameter in ADS-GAN.

Advantages of using ADS-GAN include re-identification being more or less impossible due to how the method is implemented. No parameter needs human input. All parameters are anonymized when creating synthetic data. Due to these factors, there is no risk that the developer forgets to anonymize something.

A drawback of using this method is that it has to be fine-tuned. A model has to be trained. It might not work for relational databases. It is also unclear if ADS-GAN-generated synthetic data can successfully refer to other rows of data. Foreign keys will probably not work in synthetic data. The model has to be re-trained if new data that we want to include is added or if the data changes in some way that we want to represent. Re-training the model will probably lead to inconsistent data every time. Each model will generate unique data, meaning there will be no persistence in the generated data.

4.3.1 Method complexity

In theory, implementing a machine learning method for generating synthetic data can mean that changes in the dataset, such as additions of new columns, might be a non-issue for the generation method since the network does the hard work of understanding the anonymization and the generation of a similar dataset without the operator's input. In other words, human effort does not have to be put into consid-

ering how important certain columns of data are for identifiability, which means that the risk of human error is removed from the equation, assuming the model and its in-parameters are good. The ADS-GAN with a proper identifiability property will generate properly non-identifiable data with a scalable number of data columns.

This presumes that one can get the network to work with the data we are trying to generate. In addition, fine-tuning the solution might require considerable work.

4.3.2 Relational database compatability

From a technical point of view, it is unclear if this method can be directly used to generate synthetic data for a relational database since the datasets presented by Yoon *et al.* [4] imply that the data contains homogeneous rows of simple patient data. Relational databases have multiple tables which have references to different keys in other tables.

Having one GAN for each table might work to generate the rows for that table, but having the different tables be connected using this solution would not work out of the box. The resulting data would most likely not be connected, but a bunch of unrelated rows not resulting in any hierarchy, if there is nothing to connect the foreign keys in the rows to relate to the other tables.

5 Anonymization Project Requirements

In this chapter, we will discuss the specific requirements for our specific use case. We will discuss both functional and non-functional requirements. The non-functional requirements might be less strict than those in many of the sources used in this thesis and, thus, some of them will have to be parsed from the perspective of our specific use case. In the following chapters, we will use these requirements to compare the existing methods from the previous chapter and then implement one or two solutions to achieve the goals described in this chapter.

We will need to develop a tool to provide useful data for a test or staging environment with the audience being mostly internal developers. We will have to figure out what kinds of features our solution should support, and what is possible. Having numerous vague requirements can make coming to a perfect solution difficult. Defining proper requirements and their importance before starting work on an implementation can be beneficial, especially if some requirements cause other requirements to suffer, or to be impossible.

We will have to study the benefits and disadvantages of implementing specific requirements and see how they conflict with identifiability. We might want to have a feature where the data generated is static or we might want the data to represent what our production data looks like. Will it be possible to do both using a method that at the same time provides unidentifiable data and keeps the dataset usable and relatively unchanging while also updating when the production dataset expands? Will we encounter problems if the developer looking at our data knows that a new big dataset is added to production, and the developer notices that the staging environment receives a new big dataset?

The most important requirement is to have some level of anonymization. Making the resulting data at least not a direct copy of production can be a good first step. The required level of non-identifiability has to be discussed. There are many tools and methods to measure the level of anonymity in an anonymized dataset, and those

will be discussed in various chapters.

Many of our literature sources [1, 2] in this thesis discuss anonymization with the goal of being able to release the resulting anonymized data to the public. This means that the pool of attackers and the ease of attack on the anonymized dataset or its access level will be lower than that which has been mentioned in the sources.

We want the dataset to be good for noticing possible bugs before releasing features to production. This can mean many things, such as the time it takes to search or create data.

We probably do not want to have to spend much time maintaining the tool, if our data changes for example. This means that if a solution requires a considerable amount of extra work from developers to be able to test new features, the utility of the tool will suffer. One example of this could be strict input validations. If we are, for example, anonymizing new data and there is a version mismatch, since we are deploying new features, we do not want the system to break down.

The practical part of this project will be implemented using the programming language TypeScript and will be deployed as an AWS lambda, from where we will generate an anonymized dataset based on the production environment to the staging environment.

The resulting dataset must be valid according to possible validation tools. The data must have functioning foreign keys, assuming the foreign keys point to existing rows of data in the original production data. This means, for example, that simply randomizing might not work. This will be discussed further in the next chapter.

6 Comparison

In this chapter, we will review what we learned in the previous chapters and consider what are the pros and cons of each method.

6.1 µ-Argus

The µ-Argus method is interesting. It brings many advantages, but according to for example Bergeat *et al.* [1] and Chou [3], it requires the user to input the data as a microdata file. This requirement can be good for institutional anonymizers who already have their unanonymized data in that format, but for anyone with another data format, this will cause extra steps.

The method is interesting as an example of what small things one can do to data to improve the level of anonymization without losing too much utility. An example of this would be the rounding, addition of noise, and data swapping. In addition to these, µ-Argus also has an in-built system to analyze the risk of re-identification from the anonymized data.

Even if we were not to use µ-Argus itself, we can implement several of the anonymizations steps ourselves. If we had a converter for our data to make it into microdata files, we might even use the re-identification risk analyzer to get a measurement of how well we have anonymized our data.

6.2 Token-based hashing anonymization

Token-based hashing anonymization is also, for the purposes of this thesis, an interesting method, even if it has many issues. Kumar *et al.* [2] proved it to not be a secure method if an attacker has statistical knowledge of the unanonymized data, and if the data has high levels of co-occurrences. They wrote that they recommend k-anonymous methods over token-based hashing anonymization.

This type of hashing should not be used to protect user data [15], at least not alone. Hashing algorithms should not be seen as a method to anonymize, but rather

as a method to pseudonymize. Pseudonymization means that the data can be converted to identifiable data with a key. [15]

Marx *et al.* [15] described how cryptographic hashing algorithms are a good pseudonymization method due to their one-way conversion, meaning that the output can not be used to directly get the input. On the other hand, they wrote about how personal information, such as IPv4 addresses, MAC addresses or generic e-mail addresses, and other similarly limited input sets (small pre-image space, meaning the possible inputs that lead to the same hash), can be de-hashed within hours on consumer-grade hardware.

With these two sources, we know we can not use hashing algorithms for data with a small pre-image space and high co-occurrences. This limits the usefulness of this method. In addition to the two sources cited, hashing algorithms have been extensively researched over the years to determine their level of security. More recently, emerging technologies such as quantum computing have significantly reduced the time complexity of cracking hashed values [17].

When it comes to combining hashing with other methods, Ali and Dyo [18] wrote about bucketing of hashed MAC addresses. This bucketing of hashed values can be used to reach k-anonymity in the output data if the collision rate of the output is sufficient. If there is a limited amount of outputs and a sufficiently large amount of inputs, similar to the birthday paradox, there will be ambiguity over the original value for a specific anonymized output. In the birthday paradox, Ali and Dyo [18] wrote about reaching an acceptable collision rate for their specific use case, which for them was 1%. For our use case, we might want to accept a higher percentage. In our use case, a higher collision rate might mean searches returning more results in the anonymized set than in the original non-anonymized set.

In a case where the production data set is growing, this higher collision rate might be an acceptable side-effect since, in theory, we will be getting on average more results in our test environment than in production. However, getting the results will take a longer time to execute to completion. Since this will happen in a growing data set, this can be a positive factor as a slight future predictor of how long similar searches

will take to execute in the future.

6.3 Generative Adversarial Network solution

ADS-GAN, as described by Yoon *et al.* [4], seems like a very good method to use for use cases where it can be applied. Although implementing it might be difficult, it is most likely the best method to anonymize data, since it synthesizes entirely new data based on the existing data.

In the article written by Yoon *et al.* [4], there is no mention of their ADS-GAN method working for a relational database. If the ADS-GAN method cannot be used for relational databases, the utility of it will diminish, since many databases are built using a relational database model, and they can be difficult to anonymize using synthetic methods [19].

Xu and Veeramachaneni [19] concluded in their 2018 paper about using GANs for generating synthetic data that it is difficult to use the various GAN methods to synthesize data for relational databases. They noted that their GAN method works for a database with a single table, indicating that a multi-table relational database would be more difficult to model using a GAN. Xu and Veeramchaneni [19], in turn, observed that GANs are a scalable method for keeping correlations from the original dataset to the synthesized dataset.

From a technical point of view, it seems that this method cannot be directly used to generate synthetic data for our specific use case. The datasets presented by Yoon *et al.* [4] imply that the data contains homogeneous rows of patient data.

Our use case would require the generation of multiple tables of connected relational data. In theory, having one GAN for each table might work to generate the data, but having the different tables be related to each other using separate GANs would most likely not work out of the box. Moreover, it would seem that GANs are more suitable for single-table databases. Hence, this approach, while interesting, cannot directly be applied to our relational database model.

7 Implementation

Having taken into account the different aspects of the methods we have analyzed, we can decide on what we can do with our data. In this chapter, we will first, on a higher level, consider which choices we can make to have a *minimum viable product* (MVP). After that, we can start to implement an MVP as a proof of concept. In the next chapter, we will reflect on what we have learned during the process.

An MVP implementation will demonstrate that we can automate a process to generate useful data for a test database that is GDPR-compliant and anonymized. This means that for an MVP implementation, we will first choose a specific part of our data model in such a manner that we can get a usable result with a minimal amount of anonymization steps.

For the level of anonymization, we will think about our requirements on how safe we need the data to be. Though, this has to be in balance with the level of usability. As mentioned in section 3.2, if we want the most anonymous output we can just remove everything. This means we have to weigh the utility of specific data with how much of a risk keeping its utility will be.

As a start, we will lay out the work plan so we have an overview what we need and what choices need to be made. The implementation steps are listed below.

7.0.1 Implementation steps

1. **Analysis:** First, we will need to go through a number of fields and tables and identify if the field can contain content that is in scope for GDPR. For fields that are not fulfilling the requirement, we can discuss the benefits and disadvantages of leaving them unanonymized.

   To begin with, we will start with doing this process for two tables, where the first one (Tab-1) does not have any references to other tables, and the other table (Tab-2) contains references to this first table. Other for now irrelevant tables (Tab-3, Tab-4...) are often connected to Tab-(n-1) (See Figure 2). We chose to do this for two tables to prove that our method can be used for

relational databases.

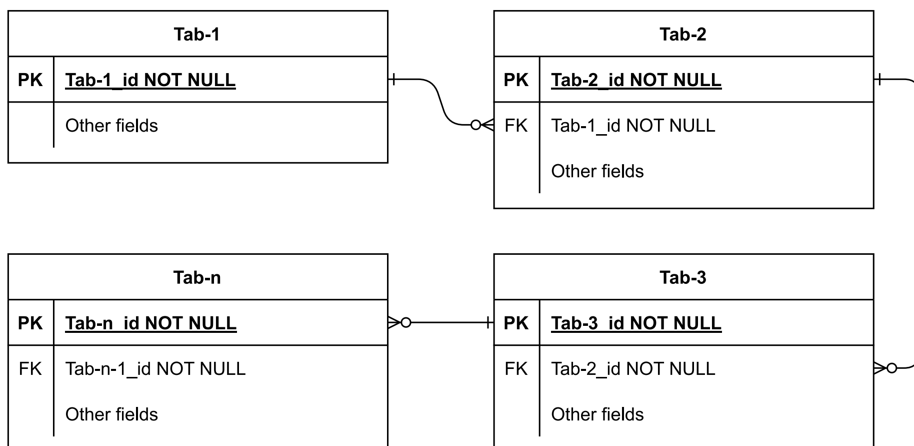

Figure 2: In this figure we can see an abstract representation of the described Entity Relationship Diagram showing the Primary key (PK) and Foreign key (FK) in each table

2. **Triggering and fetching data:** We will implement a Lambda function in Amazon Web Services (AWS) that is triggered on a daily schedule to keep the stage database up to date, and clean from possible manual changes that developers, tests or faulty code might have caused. The benefit of keeping it up to date, for example daily, is that our tests, code and developers will be trying to make their code work on an environment that is as similar to production as possible.

   When our Lambda function is triggered, it will read production data, either all rows of the two tables, or just a limited amount of rows from each table, depending on anonymization related choices we make. These anonymization related choices have to do with if we want to do "*binning*", meaning combining random rows in Tab-1 together to make it, for example, harder to know for sure which row in Tab-1 is which by knowing how many rows in Tab-2 is related to it, or which item in Tab-2 is which based on how many related items in Tab-3 it has, and so on (See Figure 3).

   For this *binning* method, to keep the resulting anonymized data similar enough to production, an equation similar to that used for the Birthday problem could

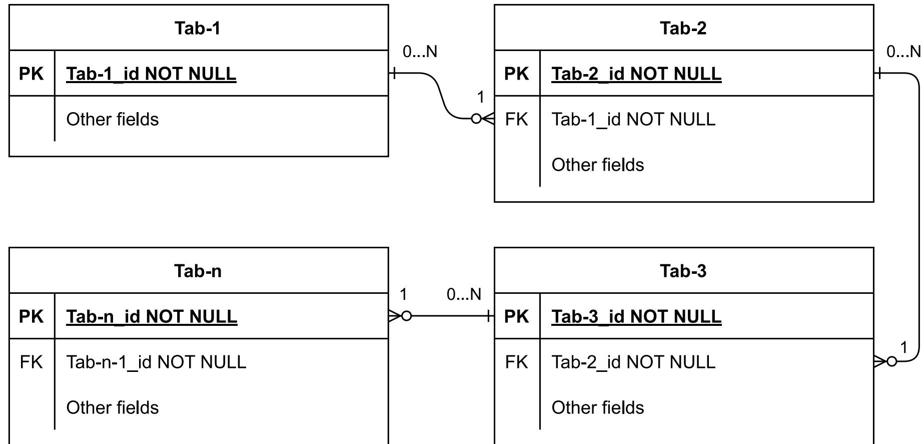

Figure 3: Adding to Figure 2, we can now see the cardinality and optionality of the relations.

be used to mathematically calculate how much binning we need to do if we want to keep sizes similar to production if we decide not to read all data from production but only a part.

3. **Applying methods:** When we have read the production data, we will have to apply the filters we decided to apply in the first step to the data. We can use Faker [20] as a helper to give us "good enough" data. Faker can be used to generate fake, but "real looking" data. Faker has many categories one can choosen from to create specific type of fake data, such as ipv4 addresses or zip codes. Using our production data we can create an input for Faker's seed feature, to keep consistency in co-occurences, but with the value not being directly identifiable.

   Since Faker essentially is a regular expression generator which results in a finite length of possible results, some *faker.seed(originalData)* $\rightarrow$ *faker.something()* can in theory result in the same value even if *originalData* is not the same, there will be some level of "binning". This is essentially equivalent to having a lossy hashing algorithm mentioned by M. Marx *et al.* [15].

4. **Keeping it clean:** When we have processed the data through an anonymization filter, it can then be inserted into either a pre-cleared stage database, or we can use an update system such as *ON CONFLICT REPLACE* which will

update each matching row of data with our newly created data to "clean" or update it. Unfortunately both of these methods have some issues that we will either have to keep as problems, or we will have to invent a more complex solution to solve them.

The issues here are that if we change how our binning algorithm works or how data gets removed from the production data, the resulting IDs might not all match what was previously produced. This can cause issues in the two scenarios.

The issue in the pre-cleared stage scheme would be if we have static tests with fixed IDs and expectations, our tests will most likely break every time we change the algorithms. The stage database will be empty until the anonymizer uploads the fresh anonymized data.

The issue in the update-based scheme would be that we would be slowly collecting a backlog of old rows. This might not be a problem if our binning limits us to a low enough amount of possible output items. Since there are a limited number of possible (binned) IDs, the old backlog items might at some point be updated if any of our newly generated items have the same ID. This also means that in theory there can be as many items in the database as we have as our binning limit for the ID. This however does not take into account possible test or developer generated rows with IDs that can not be generated by the binning algorithm. These rows would never get updated by the anonymizer in this case.

A possible solution might be to peak at each row in the anonymized dataset, after uploading or updating the fresh anonymized data. Each peaked row would be compared with our newly added and updated item that our Lambda function might still have in memory and delete everything that did not match. The issue here would be that large databases might not fit in memory, and comparing two long lists in that way most likely will be very slow: $O(n^2)$. Alternatively, if the database we are using has a "last updated" field or feature

for each row, we could just clear values that were not updated since we last updated or added new data. Otherwise if the uploading of new data can be done fast enough, the chance of this being an issue can be quite low.

7.1 Analysis

As we have chosen to limit our initial work on only Tab-1 and Tab-2 (See Figure 3), we have a smaller amount of data to analyze. In our case, as the later tables might have exponentially more data than Tab-1 for example (See Figure 3), this makes it easier for us to do a more qualitative analysis on the fields we need to analyze. In addition, limiting our analysis and implementation to only two tables leaves us with less fields compared to if we did not limit the amount of tables.

The goal in this step for us is to do an analysis of what we want to do with our data, so that we know what kind of tools we might need or what kind of system we can build to support our goals. If our requirements for example were simple enough, our system might be able to be built using less effort.

7.1.1 Goals and priorities

Our goals in this implementation, firstly, are to avoid obvious GDPR related pitfalls with fields such as “FullName” or “TelephoneNumber”, but also to obfuscate free text fields and other similarly not so obvious fields where GDPR-protected data might occur. By also censoring factors such as free text fields, we are progressing in a manner that the AOL dataset anonymizers did not, as we discussed in section 3.1.

Our secondary goal is to create a working test environment dataset, which we can use to, for example, run automated tests. We also want to keep the anonymized data relatively close to the production data, as with that, our development efforts are not focused on making the product work for something that is further from the real production data. This secondary requirement might in some cases mean that we cannot maximize anonymization by methods such as K-Anonymity, but since the data in our case will be for internal use, instead of being made for public consumption, the risk of a devious actor reverse engineering our anonymized dataset is somewhat

diminished compared to the examples in our source material. An example of this would be that in some cases where we are aware of hard-coded features, where certain values in the dataset will be tied to source code or other systems, we might have to choose to not implement anonymization of secondary identifiers. Another quality of life feature in this rank would be to have human readable data where possible, and have a similar data-size for free text fields. These requirements make it easier to work with the anonymized dataset when it comes to search-time, human readability and recognisability of fields more similar to working with production data.

Lastly, where we are not conflicting with our secondary goal, we will try to implement methods that decrease the indirect identifiability of our data. This includes obfuscating fields such as street addresses where having them unanonymized would firstly geo-locate the item, and significantly limit the possibilities of what the item can be, or more vague identifiers such as creation dates and rarely used parameters.

Rarely used parameters, or parameters where for example 99% of the values are X and 1% are Y, can reduce the anonymity of the item. If our data has multiple rarely used parameters, items which have many of these can be more easily identified. If there are several such fields, each with a majority of items sharing a common value, we can significantly narrow down the possible matches in the original dataset when several fields have a non-common value, assuming there is no correlation between these common values in the fields. Unfortunately, in some cases where we might want to change these fields to avoid this issue, we might have to prioritize the secondary requirement of prioritizing usability of the anonymized data in our use case. This would mean that some compromises will have to be made.

7.1.2 Binning

For most fields in our use case, *binning* the value should work, by using a generic value as the seed, such as the ID of the item. When binning item context specific fields, such as Tab-1.name and Tab-1.socialSecurityNumber (See Figure 4), where the context is specific to the item, using the ID or Primary Key (PK) as a seed works well, but when the context is for example a location, such as a street address

or zip code, the calculus will be different. Unlike names or social security numbers (assuming no namesakes or duplicate items), multiple items might have similar street addresses or zip codes, meaning that in a large enough dataset and in certain use cases, it might be beneficial to not just use the ID or PK of the item as the seed. It might make practical sense to, for example, use the zip code as the seed, meaning all items that share a zip code in the original data set also share it in the outputted anonymized data set.

If we want to anonymize something such as location data, where we for example have different variables such as street addresses, zip codes, city data and country codes, we can chose a different seed value for these if we wanted to group them. Reducing accuracy is good for anonymity, since that increases K-anonymity. We can then choose to bin the location fields by using, for example, the zip code, city or country code as the seed, depending on what our data looks like and our anonymization needs. In our case we will choose the city name as the seed as a demonstration. Assuming we build our anonymization code using good coding practices, it should be easy to change this in the future if the requirements change. (See Figure 4)

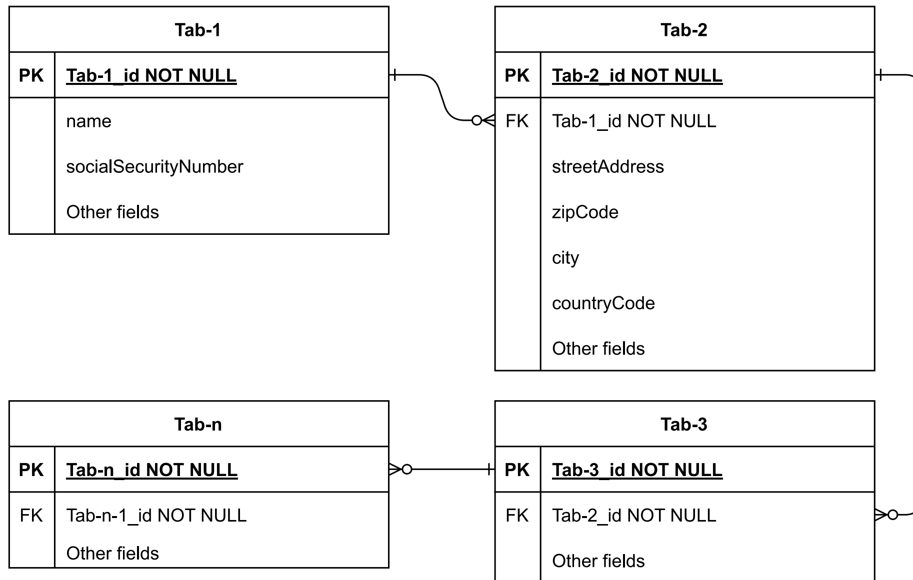

Figure 4: Figure 3 with explicit fields mentioned for Tab-1 and Tab-2

In some data sets, some tables might contain fields, where most items have the same value, and only a few differ from the most common value, and they may also

differ from each other. One example could be country codes. Consider an example where our data set contains data over European persons. If one of our fields is the country where the person lives, we can assume most of these fields will consist of the more populated countries in Europe, while a minority will consist of smaller nations. If we were to randomize each country code to any other country code, by for example *binning*, we risk making it possible to identify the resulting country code by size. In addition, very small countries can have a very small amount of occurrences. This combined with something such as The Vatican city, where there only exists one zip code, in for example a known complete dataset over the European population, any *binned* country which consists of only one zip code can be assumed to be the Vatican city. In a more generalized sense, keeping country codes can be a risk to the anonymity of the data, especially when combined with other anomalies and especially when an attacker has some knowledge over what kind of data set they are looking at. [21]

7.1.3 Noise for numbers and dates

Date and amount fields will not be anonymized using Faker-js. Rather, we will add an amount of noise to them. The amount of noise will have to be tied to the amount of noise that is needed depending on the data spread for that field. The goal is to make it difficult or impossible to distinguish items based on their exact date or amount. Another factor is clustering. If an item has a date field which is significantly far from the date fields of other items, it might be easier to spot and possibly be identified if an approximate production date is known for that item. The same would apply to amount values. This means that the amount of noise should ideally be large enough so that in theory, this kind of identification should be harder or impossible. This anonymization method is inspiered by the description of how µ-Argus applies post-randomization or addition of noise on microdata fed to it [1].

An interesting possible issue with adding noise to dates could be if we have a system where a date field in Tab-1 normally has an earlier date compared to each child [22] (Tab-2, Tab-3, ..., Tab-n). If we add noise to all dates, the noise might

cause these dates to get mixed up in such a manner that the date field in an item in Tab-2 relating to a date field in an item in Tab-1 has an earlier date than the date field in the related item in Tab-1. If this is relevant for the test environment, one solution for this could be to fix the seed for the random noise for all related fields to for example the PK of Tab-1, assuming all related date fields containing tables have a reference to the PK of Tab-1. Unfortunately, in our example data (See Figure 4), this work-around would only work for two tables, since the only shared values between related items in different tables are the FK and the PK, with the FK for Tab-n being the same as the PK for Tab-n-1. The noise modifier would be different for different seeds. The multiplier for the noise would be set to the cluster gap size. The cluster gap size (time) is the largest gap between items. In our example, each consecutive table has larger data amounts, meaning the gap is most likely smaller. If the multiplier for the noise were to be larger for Tab-n than for Tab-n-1, that would also mean this solution would not work. If we write the code to make the noise be either a positive or negative modifier, having different multipliers on the noise would not work either, since assuming positive noise and two close original dates, the item with a lower multiplier will get overpassed by the one with the larger multiplier. We figured this out by writing an unit test (See Code 9). This is further discussed in section 7.4.1.

Free text fields such as comment fields or similar are problematic since they might contain anything, and anything contains data which can be in scope for GDPR. Thus content in free text fields should not be shown in the anonymized data set. If we have, for example, a comment field, which can have any text, we might want the resulting size to be similar to the original comment to keep the size of the data similar to production.

7.2 Triggering and fetching data

We start off by making an MVP version which can be triggered manually from AWS console or automatically with an AWS trigger which is set to run every midnight. Using automatic deployments helped in debugging.

Assuming a large enough data set, one AWS Lambda might not be good enough for running this task. Theoretically there would be ways to work around this, such as splitting the task into multiple smaller tasks. This method is tailored to work in that scenario, since each item is anonymized without knowledge of items in its own table or items in different tables.

To fetch data we wrote a simple Structured query language (SQL) statement that fetches the data from an SQL database. To begin with we are just fetching data that exist in the environment that the program is deployed to. In this case, it is a test environment with a small dataset of about 75 items in Tab-1 and 106 in Tab-2. Only deploying the Lambda to this test environment protects us from accidentally tampering with production data, or from causing extra load during development.

Since deploying new code to AWS in our case takes a while, trying out our code on synthetic data and unit tests ended up being a more efficient way of testing the methods in the next section.

7.3 Applying methods

In this section we will discuss the practical parts of implementing the methods including issues and steps taken to solve them in chronological order. We will test and discuss different iterations of the program with iterative improvements or compromises to improve aspects of the program. These steps will be split into different smaller sections for improved readability. The libraries we use include faker-js 8.0.2 [20] and seedrandom 3.0.5 [23].

7.3.1 Basic binning method

As mentioned earlier, our goal is to anonymize our data and to keep the data usable and representative of the original data set. For that we will need to anonymize the data while keeping the relational structure of the data, as well as keep the anonymized values close to the original values.

Firstly, the PK of Tab-1 happens to be in the form of a short string, which also happens to be a shortened version of the name of the item. Ignoring length

```
1
2  //...
3
4  //SQL query for tab-1 ->.forEach((item) => anonymizeTab1(mapForTab1, item))
5
6  function anonymizeTab1(items: Map<string, object>, item: Item) {
7    item['tab-1_id'] = binningMethod(item['tab-1_id'], item['tab-1_id'], 50);
8    items.set(`c:${item['tab_id']}`, { channel: item });
9  }
10
11 function binningMethod(anonymizedRowElement: string, anonymizedRowElement2: string,
       accuracy: number): string {
12   faker.seed(binaryToInteger(stringToBinary(anonymizedRowElement2)) % accuracy);
13   return faker.company.name().toLowerCase();
14 }
15
16 function stringToBinary(str: string): string {
17   return str
18     .split('')
19     .map((char) => char.charCodeAt(0).toString(2).padStart(8, '0'))
20     .join('');
21 }
22
23 function binaryToInteger(binaryStr: string): number {
24   return parseInt(binaryStr, 2);
25 }
```

Code 1: Basic binning method

constraints for now, we can start testing our faker method. Some possible issues might be the chance of co-occurrence of the same output value with different seed values being too high. For that, we have conducted some tests and will display some data over the results, when using the *faker.company.name()* method.

In Code 1, we have abstracted away the irrelevant parts for this section so we can focus on the important parts relevant to the binning method. Reading from top to bottom, we are calling the *anonymizeTab1* function in sequence with each item we get from the SQL call. We give the function a map where we can save our anonymized items for later use. We can then use this map to test attributes such as the resulting size of the anonymized output compared to the original size.

In the *anonymizeTab1* function, we pass the PK (in this case, item['tab-1_id']) to the *binningMethod* function along with an accuracy variable. The PK is sent twice in this case, but that is not relevant yet. The idea is that the first variable is what we are anonymizing and the second parameter is the seed for Faker. The third parameter here is our accuracy parameter. We conducted some tests where we varied the accuracy parameter. Since the binning method performs binning even

without the accuracy variable, the variable might not be necessary depending on the targeted amount of repeated output values. We might not need to bin it in this manner.

The *binningMethod* function converts the seed into a number that *faker.seed()* can use. Since it turned out the *stringToBinary* and *binaryToInteger* functions ended up being an issue for our implementation, we will not cover them in detail. Here we used the accuracy variable to maximize the seed to a specific max seed number. If we were to compare this to the birthday paradox, the accuracy variable would be the amount of days in a year, and the original data set is the amount of people in the setting. When the modulo wraps around the seed number can be compared to people being born the next year and thus sharing birthdays.

7.3.2 Binning accuracy

To test the efficacy or necessity of an accuracy variable we ran a set of tests (See Table 1) using the basic binning code (See Code 1) on Tab-1 and Tab-2 in the test environment. The goal was to get a general idea how well our *binningMethod* works for binning using different accuracy values.

An interesting anomaly in the data (See Table 1) happens when in Tab-1 with an accuracy value of 100 resulted with a smaller resulting size than with the accuracy variable at 75. This is likely due to the modulo of some seed values wrapping to values with the same resulting value. This tells us we might have to conducts tests at a larger scale to get better test results.

From further testing it turned out that the way we parse the seed into an integer or a TypeScript number ended up being problematic. We ended up fixing this by using a cryptographic hashing algorithm on our seed when we convert it from a string to a number (See Code 2). The reason a cryptographic hashing algorithm works better here is due to the avalanche effect they employ. The avalanche effect means that if the original value changes slightly, the entire resulting hash changes [24]. This allows us to take the hash and more often get unique results for our seed.

We conducted further tests on synthetic data. First we test the original code

Table 1: The values show the resulting size of the binned (anonymized) side. The last two rows are without the accuracy variable and without the faker.company.name() parts respectively.

	Tab-1 result size	accuracy	Tab-2 result size	accuracy
original data	75	% of 75	106	% of 106
50	38	51 %	44	42 %
75	46	61 %	58	55 %
100	45	60 %	59	56 %
200	53	71 %	76	72 %
infinity	69	92 %	102	96 %
without faker	75	100 %	106	100 %

```
1 function stringToUnsignedInt(input: string): number {
2   const hash = createHash('sha256');
3   hash.update(input);
4   // Take the first 4 bytes (32 bits) of the hashBuffer and interpret it as an
      unsigned 32-bit integer
5   return hash.digest().readUInt32BE(0);
6 }
```

Code 2: A better function to convert strings to integer

without the cryptographic method. The resulting percentage of unique results in the output dataset when running the *binningMethod* can be seen in the second column of Table 2. The results for the code in Code 2 can be seen in the third column in Table 2. We generated these values up to accuracies above 50%. This test was conducted on a binning method bottlenecked by a *binaryToInteger(stringToBinary(faker.company.name()))* wrapped output. When using a cryptographic hashing algorithm (*stringToUnsignedInt()*) the amount of unique values was significantly higher.

7.3.3 Testing speed and accuracy

In Code 3 we have an example of a unit test (using the Jest test suite) used to generate the test data. We wrote two tests to assess the accuracy and speed when we are converting the *faker.company.name()* result to an integer, as well as when we are not.

Considering that for some of the Tab-n tables, we might have around 2^{24} rows of data, we might encounter performance problems at some point. Even if we were

Table 2: Comparison of the accuracy of automated tests run locally on synthetic seed values of 1, 2, 3, ... n with and without the bottleneck. The second and third columns display the relative size of the resulting data set compared to the original size for each test case.

Test Size	%: With Bottleneck	%: Without Bottleneck
2	100.00	100.00
4	100.00	100.00
8	100.00	100.00
16	100.00	100.00
32	100.00	100.00
64	100.00	100.00
128	99.22	99.22
256	98.83	99.61
512	96.29	98.24
1 024	91.89	96.68
2 048	85.40	93.95
4 096	75.07	89.62
8 192	59.80	83.85
16384	-	77.63
32768	-	71.91
65536	-	68.14
131072	-	65.10
262144	-	61.78
524288	-	56.91
1048576	-	50.35

to parallelize the Lambda or have multiple Lambdas processing the data, we would still have to pay real money for every millisecond the Lambda or Lambdas run for. This means that improving execution speed or efficiency has an effect on processing costs and time spent on waiting.

When running these tests, we can see how long it takes to run the simulated tests. On modern consumer hardware, running the tests shown in Code 3 using *faker.numbers.integer()* to generate integers for numbered IDs, it ended up taking 54 minutes and 59 seconds for the test to complete. Testing another library for generating random numbers, seedrandom ended up being significantly faster with the same test, using seedrandom (See Code 4), completing in 1 minute and 50 seconds.

```
describe('max amount at 50%', () => {
  it('binningMethod numbers', async () => {
    const tenKTest = new Map<string, object>();
    let j = 0;
    for (let i = 1; i < 14; i++) { //run 2^13 times, other test runs 2^20 times.
      const testAmount = 2 ** i;
      for (; j < testAmount; j++) {
        const row = binningMethod(123, j); //Other test has binningMethod("asd", ""+
            j)
        tenKTest.set(`m:${row}`, { fakedata: row });
      }
      const percentage = (tenKTest.size / testAmount) * 100;
      console.log('a:' + testAmount + ' %:' + percentage);
      expect(percentage > 50).toBeTruthy();
    }
  });
    //Similar test omitted
});
```

Code 3: accuracy performance test

```
const rng = seedrandom('' + stringToUnsignedInt('' + seed));
return Math.floor(rng() * (9223372036854775807 + 1)); //about 50 times faster than
    faker.number.integer().
```

Code 4: Faster seeded numbers with seedrandom

7.3.4 Improved accuracy and speed with seedrandom

The library seedrandom allows us to get a random decimal value for a given seed. It can be given a seed value as a string, similarly to how Faker can be given a seed number. For our use case, we want to convert this decimal to an unsigned integer. We do this by multiplying the decimal with the maximum number we want and adding 1 to it and then removing any trailing decimals.

When purely using seedrandom with the sequential numbers as seed values, which for seedrandom have to be string types, the results (See Table 3) for accuracy dropped from 100% down to around 93% as early as at 2^4, getting back up to above 99% at 2^{14}. First test (seedrandom) shows results when the seed is input as " + *sequentialSeed*. The second test (+ avalanche) shows the results when wrapping that with " + *avalancheMethod()*. The values in the second test had a score of 100% up as high as 2^{16} but not 2^{17}. The third test (*faker.numbers.integer*) shows the resulting accuracy using faker.numbers.integer. Results are shown in the percentage of unique generated results. When adding the avalanche effect to how we convert the sequential number seed to a string, we removed this drop, and the accuracy stayed at 100% up to 2^{16}.

Adding the avalanche step to this gives similar results as *faker.numbers.integer* does, but the results are generated significantly faster.

Table 3: This table shows the accuracy results from running the sequential seed for seedrandom test on 2^{24} values.

test size	seedrandom	+ avalanche	faker.numbers.integer
8	100.000		
16	93.750		
32	93.750		
64	92.188		
128	92.188		
256	95.703		
512	97.461		
1 024	98.145		
2 048	98.584		
4 096	98.804		
8 192	98.901		
16384	99.335		
32768	99.664		
65536	99.828	100.000	100.000
131072	99.886	99.998	99.999
262144	99.888	99.995	99.998
524288	99.890	99.993	99.994
1048576	99.895	99.988	99.987
2097152	99.948	99.975	99.976
4194304	99.974	99.950	99.950
8388608	99.987	99.901	99.902
16777216	99.989	99.804	99.806

When looking at the test results (See Table 3) we can say that for binning number IDs, using seedrandom with the avalanche effect to convert a sequential integer to a hash and turning that into a string to give as a seed to seedrandom is significantly faster than using *faker.numbers.integer*, and the results are similar.

7.3.5 Shortening strings

Now that we have methods to convert a text or a number seed into a unique human readable text, we might still have to edit the resulting output to fit better into our specific use case. A database might have restrictions such as maximum lengths for

specific columns. In our case, we sometimes have to shorten the resulting output string. Shortening allows us to have human readable text even on fields with a character limit if necessary. Other uses for this could be if there are values such as converting a *faker.company.name* result to a shorter version of the same text if the data has such fields (ie. OP or ABF versus Osuuspankki or Ålandsbanken Finland). One way of doing this could be to remove vowels and white space from the results until the size fits the criteria, and if that is not enough, more can be removed. A risk with this is that the accuracy might in some cases be lower than in our previous tests, but since we have unit tests that can test the accuracy, we will be aware of these changes if they happen.

To attempt to implement a way to fit our resulting readable string to a limited length field we can wrap it with a function that works it down. We give the function the maximum size we want the string to be along with the string itself. The goal is to keep maximal legibility and uniqueness while removing or replacing characters from it. We will call this function “shorten” (See Code 5).

Capitalizing the first letter of each word is used here to signify the start or amount of words in the string. Consonants could be considered to have more value than vowels, as in some Semitic written languages the vowels are entirely omitted [25]. Then, on top of this, if the string still is too long, rather than cutting it off to fulfill the maximum length requirement, we can remove remaining uncapitalized consonants. This system removes the vowels and respectively the consonants in sequential order. In some cases other orders of removing letters might be beneficial.

Due to the characteristics of *faker.company.name*, which generates strings such as *'feest, hintz and hettinger'* and *'ratke, grant and keebler'*, the word “and” is very common. We ran tests (See Table 4) with the full shortener, no word replacement list and only the “and” replacement. Results are shown as the percentage of unique generated results. When testing the code before adding the replacement list feature, the word “and” would be converted to “N”, since “a” is a vowel, “n” gets capitalized and since “d” is a lower case consonant, it gets removed before the string is concatenated. “N” as a shortening of “and” is not completely illegible, but since there are

```typescript
export function shorten(maxSize: number, input: string): string {
  let formattedString = input;

  //Replace known common words
  const replacements = [
    { pattern: /\band\b/gi, replacement: '&' },
    { pattern: /\byou\b/gi, replacement: 'u' },
    { pattern: /\bfor\b/gi, replacement: '4' },
    { pattern: /\bto\b/gi, replacement: '2' },
    { pattern: /\bare\b/gi, replacement: 'r' },
    { pattern: /\bat\b/gi, replacement: '@' },
  ];

  for (const { pattern, replacement } of replacements) {
    formattedString = formattedString.replace(pattern, replacement);
  }

  //Remove vowels
  while (formattedString.length > maxSize && /[AEIOUaeiou]/g.test(formattedString)) {
    formattedString = formattedString.replace(/[AEIOUaeiou]/, '');
  }

  formattedString = formattedString.toLowerCase().replace(/(^|\s)\S/g, (match) => match.toUpperCase()); //Capitalize the first letter of each word
  formattedString = formattedString.replace(/\s/g, ''); //Remove all white spaces

  //Remove uncapitalized consonants until the maximum length is reached
  while (formattedString.length > maxSize && /[bcdfghjklmnpqrstvwxyz]/.test(formattedString)) {
    formattedString = formattedString.replace(/[bcdfghjklmnpqrstvwxyz]/, '');
  }

  //If the length is still too long, trim the string to the maximum length
  if (formattedString.length > maxSize) {
    formattedString = formattedString.slice(0, maxSize);
  }

  return formattedString;
}
```

Code 5: The Shorten method

better ways to represent "and" as a single character, we might as well use it. "N" could be "nigel" or "and", so it would be better to convert "and" to "&". While we are at it, we can create a list of similar possible replacements. Unfortunately for these efforts though, *faker.company.name* only produces "and" out of our replacement list items as shown in our test results. Our example strings when input to our shorten function with a maximum size of 10 become *'F,H&Httngr'* and *'R,Gnt&Kblr'*.

Table 4: Results of testing three variations of the shortening method at max size of 10 when testing the *faker.company.name* faker feature.

test size	full word list	no words replaced	only and replaced
64	100.00	100.00	100.00
128	99.22	99.22	99.22
256	99.61	99.61	99.61
512	98.05	98.05	98.05
1 024	96.58	96.58	96.58
2 048	93.75	93.75	93.75
4 096	89.36	89.31	89.36
8 192	83.42	83.29	83.42
16384	77.01	76.78	77.01
32768	71.02	70.65	71.02
65536	66.60	65.92	66.60
131072	62.47	61.13	62.47
262144	57.63	55.33	57.63
524288	50.98	47.45	50.98
1048576	42.37	N/A	42.37

The summary for this method is that the method seems to work for the intended purpose. Depending on the use case, the desired binning rate and the size of the original data set, this method will work for generating human readable and unique replacements for real data and can be used to "bin", which decreases the re-identification risk. If the anonymization method produces too high a percentage of unique results, an accuracy variable can be added to reduce the uniqueness of the results. The variable should be a function of the total size of the original data set and the desired accuracy. This function can be calculated with the help of the mentioned unit test, or a derivation of it. An accuracy can be guaranteed for synthetic data, but the actual resulting accuracy will depend on the real original data set.

```
1 case 'city':
2   processedData[key] = binningMethod(data['city'], faker.location.city);
3   processedData['streetAddress'] = binningMethod(data['city'], faker.location.
        streetAddress);
4   processedData['zipCode'] = binningMethod(data['city'], faker.location.zipCode);
5   break;
6 case 'merchantAddress':
7 case 'merchantPostcode':
8   break;
```

Code 6: Reducing location accuracy to city level

7.3.6 Using the binning method for non-ID values

Since faker has a lot of different types of fake data it can generate, we can use it to generate fake but real looking data for other types of data, for example, phone numbers, usernames, emails, street addresses or social security numbers.

Our binning method is essentially just a wrapper to call *faker.company.name* with a set seed. In typescript, we can pass a function as a parameter, we can change its behavior to use a chosen faker method. Thus we can call it with our desired faker method as an inparameter: *binningMethod(mySeed, faker.hacker.phrase);*

Even though faker has a vast amount of faking methods, some formats are not supported. In some cases we can find faker methods that result in almost what we desire, and then manipulate the resulting string to match our requirements.

For the location accuracy part, we can call the binning method when we are handling the location accuracy variable that we want to use as the seed for the higher accuracy location variables (in this case, *streetAddress* and *zipCode*). This code (See Code 6) is run in a switch case where the key is the index of the field. In the switch case, default maps the key to itself without anonymizing, so for the higher accuracy location variables we have to break the switch case to avoid them being re-mapped to a non-anonymized value. If there is no value in keeping location data varied or semi-accurate (in consistency at a specific accuracy level) in the test environment, a PK can be used as a seed instead, to make the resulting value more random. This would decrease the risk of re-identification and might simplify the code, increasing maintainability.

For free text, we can use *faker.lorem.words*, which can be given parameters for

```
1 if (fakerMethod == faker.lorem.words && exampleValue) {
2     return faker.lorem.words(exampleValue.trim().split(' ').length);
3 }
```

Code 7: Free text field anonymization

the amount words to generate. This faker function generates us meaningless words [26]. We want to give the function the amount of words in our free text field so we can approximate the size of the field, assuming the length of the free text affects performance, we will thus attempt to keep similarity with the production data set. Unfortunately, if we want to choose the amount of words, this will not work exactly in the same way as for example *faker.hacker.phrase*. We will have to modify the *binningMethod* signature to also be able to take either the original free text or a number of desired words. We will modify the *binningMethod* to have an additional optional variable of type string. If our faker method is one that needs a length value from this example string, we will check it, take the amount of words from the example string and take that as the word amount variable.

In this code (See Code 7) we first check if we are using *faker.lorem.words*, and that we have given the *exampleValue* string. Then we trim it to remove any outer white space, and split it on ' ' or white space so we can count the length of the resulting list of words. This gives us the amount of words in the *exampleValue* string which we can then pass directly as the number-of-words parameter to *faker.lorem.words*.

The implementation in Code 7 has some issues, as the text being derived from *faker.lorem.words* is not a necessity. Using Faker here might be unnecessarily complex to fill the data-size "requirement" and there are simpler ways of doing this.

7.3.7 Random noise for date values

For adding noise to dates, we can write a method to do that for us. In our case our dates are mostly in an ISO standard (ISO 8601, e.g., 2023-11-12T18:36:06.563165Z) format. Some of our values are not given in Greenwich Mean Time (GMT), as they are missing the "Z" at the end. For our use case the time zone does not have a significant effect on the resulting data. The noise amount is large enough to make a

```
1  export function addNoiseDateTime(maxNoiseMS: number, dateString: string, seed:
      string): string {
2    const date = new Date(/Z/.test(dateString) ? dateString : dateString + 'Z');
3    if (isNaN(date.getTime())) {
4      logger.error('Invalid date string ' + dateString);
5      return '';
6    }
7    const rng = seedrandom(seed ? seed : '');
8    const modifiedDate = new Date(date.getTime() + (rng() - 0.5) * 2 * maxNoiseMS);
9    return modifiedDate.toISOString();
10 }
```

Code 8: Method to add noise to dates

plus or minus of a few hours difference to the original date value of no consequence as our *maxNoiseMS* variable is set to multiple days or even months. We can normalize missing time zones to GMT.

In code 8 we first check if the *dateString* is missing the Z, and if so, we add it to the string. This means we just assume the date is given as GMT. We then convert the string to a Date, and check if it succeeded. We use seedrandom to generate a seeded random number as we did with number IDs. The *seedrandom(mySeed)* method gives us a variable, here called *rng*. It gives us a number between 0 and 1 which we then manipulate to give us a random noise amount up to our *maxNoiseMS* variable in either negative or positive milliseconds to add to our original input date.

The code should be improved by making the anonymized output appear in the same format as the original given value. This could be important since changing the format from the original can cause issues as we are relying on new features and releases to work in our test environment. The test environment should be as similar to production as possible. If a new feature does not fail due to the noise function sanitizing it to be in a specific format and the production environment has another format, there is a risk that the new feature does not work correctly in production even if it worked well in the test environment.

A similar method could be written to add noise to amounts. The resulting method would be simpler since there would be no need to convert a string to milliseconds. A possible improvement would be to instead use Gaussian distribution instead of this. Doing that would make the data more anonymous, as there is no *maxNoiseMS* which a reverse engineer might use to extrapolate what the original date was as easily, but

it might affect the usability of the generated data.

7.4 Improving the code

In this section we will discuss how we improve our code. Increasing the code quality is done by writing unit tests to verify that all parts of the code do what they are supposed to do. We will also take a look at the performance of our code and see what we can do to decrease execution time to make it usable for larger datasets.

7.4.1 Testing and code quality

To test the program, we have implemented a number of unit tests to test the methods, as well as tests where we mock SQL searches to test the entire flow (apart from an actual database connection). The tests are run using Jest. The Jest framework has a feature where you can check the code test coverage. Using this feature the programmer can figure out which parts of the code are not covered by tests. This can help the programmer find out what they have not written tests for. When the programmer writes tests for these uncovered parts, they can often reveal bugs in the implementation.

For each step of implementing features for the anonymizer program we wrote tests to cover the changes. If changing parts of the anonymizer changes the output, we are prompted to change the expected values or rewrite the test. When writing tests, our goal was to get the test coverage to 100% to cover all programmatic branches in the code.

The value of these tests lies in the fact that if a change to the program causes any tests to fail, we can not merge that code to the master branch unless we either fix or remove the tests, or the code that makes the tests fail. Often this can involve changing the expected results of a test for a specific fixed input value, as a change in the code might change the output of the resulting anonymized output. This means the programmer can notice the resulting changes their updated code causes. This lets them consider if they are changing the code in the intended way.

```
describe('time noise', () => {
  it('should produce logical date order for Tab-2 item related to Tab-1 item', async () => {
    for (let i = 0; i < 50; i++) {
      const fk = new Faker({ locale: [en] });
      const tab1Id = fk.company.name();
      let inputTab1 = { ...tab1, tab1_id: tab1Id };
      const outTab1 = anonymizeTab1(inputTab1);
      let inputTab2 = { ...tab2Item, tab1_id: tab1Id };
      const outTab2 = anonymizeTab2(inputTab2);
      expect(new Date(outTab2.createTime).valueOf()).toBeGreaterThan(
        new Date(outTab1.createTime).valueOf(),
      );
    }
  });
});
```

Code 9: Testing that order of dates stays the same for related Tab-1 and Tab-2 items regardless of noise seed (tab1Id)

When writing tests, we start by writing simple tests for happy case scenarios, where for each tested method, we give a simple input to test that the most common use case works as intended. This usually means the code test coverage increases from zero to for example 70%. The remaining uncovered code usually includes parts of the code where we have conditionals such as not calling the binning method if a value is just an empty string.

For some features writing tests by looking at the coverage does not work. An example of this is the *addNoiseDateTime* method that was designed to make specific date values in Tab-1 and Tab-2 have the added noise not change the order of the dates as mentioned in section 7.1.3. For this, we wrote a test (See Code 9) that creates a shared id that we set as the PK for the Tab-1 item and as a FK to Tab-1 in the Tab-2 item. With the PK of Tab-1 being the seed for the noise amount, we test the anonymized values to check that the date of the Tab-2 item is a date after that of the Tab-1 item. To avoid the test giving us false positive results, we run the test a number of times. Running it multiple times takes only tens of milliseconds. In this example we use JSON templates so we can specify only relevant parts of the input data to make the test easier to read.

```typescript
const [tab1, tab2] = await Promise.all([
    getTab1(db), getTab2(db)
]);
```

Code 10: Parallel database queries in TypeScript

7.4.2 Execution time at scale

In this subsection we will run some tests to figure out how well this program would run if it were to be run at a larger scale. To do this we will have to simulate the program getting several results from the SQL queries. At this point the code (See Code 10) can read Tab-1 and Tab-2 asynchronously. Our test code can mock the results of the SQL queries to give us a specified amount of generated data, so we can simulate the program running with any number of items.

After the queries, the code calculates the amount of rows, and the anonymization functions for Tab-1 and Tab-2 are called sequentially. These functions process each item in Tab-1 and Tab-2 using our chosen anonymization methods. For each item, we process each parameter according to what we have chosen to do with it.

In our case, we have chosen not to process properties which we have not done an analysis for, as opposed to dropping them. The goal here is to keep the usability of the test data high. This means that we might risk getting a new field which could contain sensitive data which we would not have an anonymization process for. The new field would just be mapped as it is to the test data set.

After the anonymization functions have been executed, the results are saved in a list where only unique PK-containing anonymized Tab-1 and Tab-2 items exist. We have not implemented any SQL inserting systems, but instead display the results as logs. This way we can debug the results. For the future, these lists can be inserted into a specific database.

To test the large scale performance of our full program we can mock the *getTab1(db)* and *getTab1(db)* results using Faker to create unique enough items. We generate n total items, n/2 items for Tab-1 and n/2 items for Tab-2. With these unique enough items we can run the anonymization methods to measure the execution time for a large amount of items.

This test does not include the time it takes to get a result from the database. The main goal here is to get an idea how efficient the design is. Database querying time is dependent on factors outside of the scope of this thesis. For this test, each item is on average operated on 11 times by the binning method, 3 times by the shorten method, 2.5 times by the data noise adder and 0.5 times by the fast seedrandom number method. These tests were run using a Jest test.

Table 5: Total time shows time for program to execute from generating n items of input data (executed in parallel for Tab-1 and Tab-2).

n	output	accuracy	total (ms)	generate data (ms)
4	4	100 %	18.342534	2.786242
8	8	100 %	29.329461	
16	16	100 %	50.576841	
32	32	100 %	93.287598	
64	64	100 %	180.310172	8.209222
128	126	98 %	349.419097	
256	252	98 %	691.037353	
512	505	99 %	1329.622816	
1024	1017	99 %	2646.085011	
2048	1984	97 %	5097.051695	79.890492
4096	3872	95 %	10186.97936	
8192	7461	91 %	20269.66346	
16384	14115	86 %	40919.69487	489.11608
32768	26465	81 %	81889.15668	973.746218
65536	49952	76 %	160259.8462	1831.406739

The accuracy values for these these tests (See Table 5) are lower than expected on the basis of the "synthetic" tests, since the seed value here is generated by running *faker.company.name()* instead of sequential numbers. Looking at the time the tests took to execute, they are significantly slower than we would expect from the synthetic tests. After some profiling of the code, the biggest cause of the slowness was identified in the binning method. Further analysis showed that executing *faker.seed* in the binning method took around 0.2 milliseconds, which is a significant portion of the per-item execution time when we consider the tests give an average execution time per n items of about 2.49 milliseconds and that *faker.seed* is called 11 times for each item on average.

7.4.3 Faster performance without seeds

Setting our performance goal to be able to execute this program with a data set containing around 2^{24} items, extrapolating the data from our table as a linearly increasing time complexity, at 2^{24} items we would have to spend over 11 hours of CPU time for the program to complete. Even if we were to parallelize the anonymization steps, it would still be computationally expensive, even if the execution speed would in terms of time be shorter.

If we do not care about consistency between runs for diverse fields we can stop using *faker.seed* in the binning method. Then we could get an almost two orders of magnitude faster execution speed for our test. It is worth noting that without a seed for PKs and FKs, the relational features of our anonymized data will not have a structure resembling the original data, but will instead be random, similarly to the birthday problem.

Table 6: progressively reducing code complexity step by step and measuring execution time with n = 65536 using the same parameters as in the previous table.

Change	output	total time (ms)	data generation (ms)
no change	49952	160259.8462	1831.406739
no faker.seed	55948	9158.370621	1855.592549
2x faker.string.uuid to randomUUID	55889	8288.529521	1882.717965
1x seedrandom to Math.random	55822	7251.870718	1898.894544
no shorten in non ID fields	56005	6820.11642	1910.856225

Reducing the complexity of the program by removing usage of *faker.seed* seems to have made the simulated test (See Table 6) run about 17 times as fast as the version which used *faker.seed*. Since no seed is used, the output amount varies, but the accuracies round to about 85% for all the changed tests. The data generation step already uses unseeded faker and thus has some extra variation in execution speed in addition to other variation causing factors such as background tasks. Further improvements related to optimizing seed related issues gave us even faster code. The *faker.string.uuid* function was an outlier among other used Faker functions in execution speed and could be replaced with the faster *randomUUID* method from the Crypto library since we do not use *faker.seed* anymore. Following this pattern

we can also replace our seedrandom code with *Math.random* instead which also increased performance. On top of this we simplified some code where we wrapped a binning method using *faker.internet.userName* with the *shorten* method and some string manipulation used to re-create a specific username format to merely fetch the *faker.internet.username* directly. This way we simplified the code and the execution time. With all these steps we got an execution time about 23 times shorter or reduced it to about 4% of the original unoptimized code. After these optimizations we found no significant outliers causing increased time complexity other than the amount of fields we are anonymizing. Thus we reduced the extrapolated CPU time of processing 2^{24} items from about 11 hours to about 48 minutes. The tradeoff here is that the results do not remain the same between different runs of the program. Anonymity is also increased, as there is less coupling between the two datasets.

8 Discussion

In this section we will discuss different issues we had with the chosen method. These issues are categorized into subcategories. We will also shortly discuss certain implementation details which were not mentioned in enough detail in the implementation section.

8.1 Execution time related issues

We started to work implementing binning using Faker with the assumption that it could be used for almost every field while calling the *faker.seed* each time. In small scale tests it seemed to work well enough. Unfortunately, for our performance requirement of 2^{24} items with a dozen anonymization steps for each item, calling seed each time makes the program too slow.

If we wanted to run the anonymizer daily, the program should be able to execute within a reasonable time. If the anonymization performance for our required scale is beyond that amount of time, running it would be too slow and costly. The performance with all values seeded, ended up being about 11 hours (extrapolated) of single threaded execution time. This would not be as big an issue if our performance requirement was lower, or if we were to parallelize the program.

The program could be parallelized if we changed it so that it can trigger itself in AWS. This would mean that first, the nightly schedule triggers the Lambda without any parameters or with some specific parameter. This would make that instance know it is the master Lambda. It would analyze the size of the data set. Depending on the size, the program would split up the work, either by creating and sending the SQL query as a string to the slave Lambdas or by giving them pagination information (a limit and an offset variable). Those slave Lambdas would then process the data from the SQL query.

8.1.1 Faker-js

For their next version, Faker-js has defined performance increasing goals in their roadmap [20]. The performance goals have to do with modularization of the different packages. In theory there is a possibility that the issues with performance of *faker.seed* can be improved by that. On the other hand, we do not know exactly why the *faker.seed* function is so slow.

It is, however, unlikely that even with performance improvements, functions such as *faker.string.uuid* or *faker.number.integer* will be faster than using more conventional methods such as *randomUUID* or seedrandom. This is due to Faker using string handling with regular expressions to generate this data. It is simply faster to generate the data with more primitive data structures.

8.2 Database and connection related issues

What if the database keeps on changing during this execution? We can in theory copy or clone the original database, which should be relatively fast. Having a temporary database would also solve the issue with removing the last day's data from the anonymized database, as we can copy the temporary database over the anonymization database after anonymization is completed on that database.

Important here is that the copying of the temporary database happens only after all anonymization is done. The temporary database should also exist in the same environment as the production, or the original database, as the database contains production data before the anonymization has completed.

If we parallelize the program, we might have issues with allowed connections for the database. For this reason it is also important to not have the Lambda swarm spam the production database, but instead the temporary database, as that could cause issues with daily production operations.

If the production environment is heavily separated from the test environment, another issue can emerge. As with the temporary database being in the production environment, since it contains production data for the moment, it is important that

we do not give production access to a Lambda in a test environment. This means that we have to give the anonymizer in the production environment permission to connect to the test environment.

8.3 Test structure for the anonymizer

As we want to test all our code changes before pushing them to production, we would also want to test the anonymizer. To do this we would need to duplicate our database structure. At a minimum we would need to be able to test that our anonymizer works, by anonymizing some amount of data and saving it to an anonymous database. This would also require the code to be modular in the sense of us being able to define which database is used as a source for the data, and which is used as an output database. That can be done by defining them as we deploy the Lambda to AWS.

8.4 To seed or not to seed?

In theory, not using existing values as seeds to create anonymized data could be considered a better way to anonymize, since it would be harder to track changes in production data based on how much the anonymized fields change. A possible down side of this could be if the use case required the anonymizer to generate static data. Even then, if the original production data changes, the generated test data would change even if we seeded the Faker calls in the binning method.

A compromise between these two extremes can be just reducing the amount of times *faker.seed* is called. If for example we used the same seed (such as the PK) for each call of the binning method for each item, we could consider calling *faker.seed* only once (instead of 11 times). This means that the results would be the same for each item. Each item would theoretically in our case execute in about 0.49 ms meaning the code would run 5 times faster than with running *faker.seed* for each use of the binning method.

One interesting side effect of only calling *fake.seed* once per item occurs when the number of binning method calls change over time for a specific item. If a field

changes to an empty value or is omitted entirely, we do not execute the binning method to anonymize that field. The changed amount and order of fields change the increment of the random seed compared to an earlier anonymization run. All binning method calls after this ommited field will have different anonymized values.

The change in the random order when seeding only once also happens for our free text anonymization. When we call *faker.lorem.words* with a specified amount of words, n, as the parameter, the seed gets incremented n times. This is in practice similar to calling any Faker method n times. If the word amount in any free text field using this method changes, all consecutive fields will also change in the anonymized output. We ended up changing our use of *faker.lorem.words* to just multiplying the string 'words ' by the amount of words, and trimming it instead due to this inconvenience.

When not using seeds for all fields, there will be issues with creating unit tests. In some cases, such as in the free text issue, *faker.lorem.words* causes irrelevant fields to change. This makes writing tests for different cases more complex than necessary when irrelevant fields change due to non-related fields changing. For tests, we use a fixed seed which we can set before each test is called. Thus, we can hard code the expected result for a fixed input. For fields which are set by *randomUUID* we just check that the string matches the regular expression of a UUID.

As calling *faker.seed* once per item does not seem to not bring consistency as much as expected, we might as well not call it at all. Tab-1 has a PK which is a string. The consecutive tables have PKs which are unsigned integers.

Calling *faker.seed* for the PK of Tab-1 means we also have to call it for the FK in Tab-2. For Tab-1 it might make sense to call *faker.seed* with the PK of Tab-1, so that the results would be unique. What seed would the program use for Tab-2? If it was to call *faker.seed* for Tab-2 using the FK (to keep the relational features working), all items in Tab-2 which share a FK would have almost identical results. What seed would we use for the rest of the tables, which have no string based PKs or FKs? The seedrandom based solution is faster than using Faker for that purpose.

To keep the anonymized relational data more similar to the original production data, the relational structure needs to be kept similar between the two. We must, thus, keep the PKs and FKs consistent between tables, even if not between runs. As it turns out, Faker-js is not the only tool we can use. Using a cryptographic hashing function and truncating, or cutting off part of the results can allow us to achieve similar results to using Faker with a seed for the PKs and FKs. This is significantly faster than calling *faker.seed*.

9 Conclusion

In this thesis we explored the feasibility of Faker-js as an anonymization tool to generate production-like relational test data. The original goal was to use *faker.seed* to generate consistent results every time the anonymization is applied to the same data, but due to lower than expected performance, we did not use *faker.seed* in the end. Consistency, though, is necessary for fields such as primary and foreign keys. For this to work, we used other methods such as seedrandom to get faster and consistent results.

Faker is easy to work with as it allows the programmers to pick one of many modules with functions to choose what they want the anonymized data to look like. Although, the slow performance when using Faker with a large enough scale could cause issues. If performance at the desired scale is too slow, some Faker functions can be replaced by other faster counterparts or similar methods, or *faker.seed* can be omitted.

Using *faker.seed*, seedrandom or similar methods works well for anonymizing private and foreign keys for relational data, while keeping the relational structure to the one shown in Figure 2. According to Xu and Veeramachaneni [19] using emerging methods such as ADS-GANs to generate synthetic data might not work for this kind of relational data.

We believe this method is beneficial for generating usable, anonymized and production-like test data for use in a test environment. Using this anonymizer will increase our code quality and testing practices as well as reduce production bugs while keeping the separation of the test environment and unanonymized production data.

10 Anonymisering av produktionsdata för att skapa fungerande och användbar testmiljödata

I denna avhandling diskuteras anonymisering av produktionsdata för användning i testmiljö.

10.1 Introduktion

I många källor [1, 2, 3] diskuteras anonymisering av produktionsdata för offentligt bruk. Det problem som ska lösas är problemet med dåliga testdata i vår testomgivning. Dåliga testdata kan orsaka problem inom programutvecklingen. Om programmet utvecklas med felaktiga data kan utvecklaren anta att programmet kommer att fungera, även om produktionsdata kan se annorlunda ut. Det kan handla om att produktionsdata innehåller data med annan formatering, eller att datamängden är annorlunda i produktionsdatabasen. Om datamängden skiljer sig kan söktiderna öka för databasanrop. Detta kan resultera i antingen oförväntad långsamhet eller i att sökningen helt enkelt misslyckas för produktionsanvändare.

Idén är att om testdata önskas vara mer lika produktiondata, skulle produktiondata kunna kopieras till testmiljön. Problemet här är att om produktionsdata innehåller personuppgifter kan detta innebära problem med hantering av dem.

10.2 Laglighet

Lagen som mest berörs är den allmänna dataskyddsförordningen, eller GDPR. GDPR innebär att personuppgifter inte får spridas eller dupliceras utan att först anonymisera dem [7, 4]. Persondata betyder data som identifierar en person direkt eller indirekt. Enligt Yoon *et al.* [4] är det oklart vad anonymisering i praktiken betyder. Dessa krav tyder på att gränsen för vad anonymisering betyder inte är starkt definierad. Alla data som kan innehålla direkt eller indirekt identifierbara data måste därför anonymiseras.

10.3 Perspektiv på anonymisering

Nedan presenteras några exempel som är väsentliga för anonymisering. Ett exempel är America Online-datasläppet år 2006. America Online (AOL) tog en delmängd av sina sökmotordata deras användare hade skapat. AOL hade anonymiserat användarnamnen genom att hasha dem [2]. Det som AOL inte hade anonymiserat i sina data var själva sökmotorsökningarna. Sökmotorsökningar är textfält där användaren kan skriva vad som helst. Forskare kunde använda de oanonymiserade datan för att identifiera personer genom att granska användarens sökningar efter restauranger i deras hemstad och släktnamnssökningar. Risken i AOL-exemplet var att man kunde identifiera individer vars data var anonymiserade, baserat på bland annat fritextfält som innehöll identifierande och känslig information.

Ett annat exempel från 2006 är händelserna kring Netflix-data. Netflix släppte ut data där de också hade anonymiserat användarnamn. Problemet med datan var att om någon hade förkunskap om mer unika datapunkter, kunde de koppla samman data med användare [9]. Det här kan kallas k-anonymitet. Begreppet k-anonymitet betyder att datan man behandlar har minst k stycken liknande datapunkter. Som ett exempel kan nämnas att om k stycken användare har samma data och deras andra direkt identifierande data har anonymiserats, kan man inte direkt lista ut vilken av de k stycken människorna en viss k-anonymiserad datapunkt handlar om.

Sista exemplet i detta kapitel är sociogram. I och med att företag såsom Facebook har vuxit, har forskning på sociogram utförts [12]. Sociogram skapas genom att man observerar datastrukturen som bildas från nätverk av alla kontakter och användare på sociala medier eller dylikt. Sociogram uppstår av personer (noder) och deras kontakter (kanter). En nod kan ha en viss mängd kanter, och grannarna till någon nod därtill ha en viss mängd kanter. Grannarna kan också ha gemensamma grannar vilket kan vidare analyseras eller användas till identifiering. Då det kommer till anonymisering av sociogram kan det vara svårt att fördunkla dessa faktorer om man vill behålla datans användbarhet. Även om man anonymiserar faktorer i noderna (direkt identifierande data) kan man ändå identifiera noder baserat på deras grannskapsgraf genom att jämföra den med de oanonymiserade datan.

Beroende på vad man vill använda data till kan man göra vissa kompromisser med dataintegriteten i samband med anonymisering utan att förlora användbarhet.

10.4 Anonymiseringsmetoder

För att veta vad andra anonymiseringsprojekt eller verktyg gör kan man analysera dem. För avhandlingen analyserades en mängd verktyg och anonymiseringssätt för att senare analysera om de kommer att fungera i det aktuella fallet.

Ett europeiskt verktyg, µ-Argus, är ett verktyg byggt för att anonymisera mikrodata. Mikrodata är en datatyp som används av institutioner och folkräkningsdata. Med µ-Argus kan en anonymiserare importera sina mikrodata för att köra anonymisering. Anonymiseraren väljer några parametrar för sina datakolumner och µ-Argus anonymiserar data genom att bland annat lägga till brus och konkatenera flytande nummerfält. Eftersom µ-Argus är byggt för mikrodata är det inte direkt tillämpbart i vårt användningsfall. [1, 3]

Tokenbaserade hashningsalgoritmer är en metod som också har använts för att anonymisera data. Det finns dock problem med tokenbaserade hashningsalgoritmer som anonymiseringsmetod. Många hashningsalgoritmer kan bli knäckta med relativt lätt hårdvara och relativt kort tid. Knäckningsmetoden som Kumar *et al.* [2] beskriver kräver att angriparen har statistisk kunskap om data och att data har en hög nivå av samstämmighet.

Maskininlärningsmetoder som ADS-GAN har använts för att generera syntetiska data. Yoon *et al.* [4] experimenterade med en modifierad ADS-GAN, eller "Adversarial Generative Adversarial Network". ADS-GAN är en populär maskininlärningsmetod där två skilda nätverk samverkar. Det ena nätverket lär sig att skapa datan som liknar de ursprungliga data, medan det andra avvisar data som liknar det ursprungliga utbudet för mycket. Med denna metod måste man finjustera hur liknande data man tillåter. Man måste också träna modellen om man vill synkronisera utbudet att likna produktionsdatans tillstånd.

10.5 Projektkrav

Projektkraven är att ha ett system som kan anonymisera produktionsdata för bruk i en testmiljö. De anonymiserade datan måste vara giltiga data som fungerar enligt förväntningarna i testmiljön. Om data inte fungerar på samma sätt som i produktionsmiljön kommer de inte vara till nytta. Samtidigt får inte de anonymiserade datan innehålla persondata. Det önskas inte heller behöva lägga onödigt mycket tid på att underhålla anonymiseringssystemet. Anonymiseringsprogrammet önskas skrivas med TypeScript. Datan befinner sig i en relationsdatabas och tabellerna är sammankopplade genom primära och sekundära nycklar.

10.6 Jämförelse av metoder

Efter att projektkraven har tagits upp kan det analyseras hur bra de tidigare metoderna passar till dessa krav. Även om en given metod inte kan användas direkt i implementationen, kan vissa delar av den tas som inspiration till vad som kan göras.

Anonymiseringsverktyget µ-Argus kan inte direkt användas i det här användningsfallet, då datan inte är mikrodata. Det intressanta med µ-Argus är sättet det hanterar siffror. Konkatenering och buller är metoder som kan implementeras utan att använda µ-Argus.

Hashing är lite svårare att andvända säkert, men om hashing kombinerars med andra metoder kan den i teorin användas. Hashing är i princip en enkelriktad algoritm som kan ändra ett ingångsvärde till ett annat på ett sådant sätt att det är svårt att hitta det ursprungliga ingångsvärdet med hjälp av utgångsvärdet. Nyckelordet är destruktivt. Vissa hashingalgoritmer är destruktiva, vilket innebär att med data man får ut ur algoritmen kan man inte direkt få ut det ursprungliga värdet. För varje destruktivt hashade värde finns det en hel mängd möjliga ursprungliga värden. Beroende på hur destruktiv algoritmen är, kan mängden möjliga ursprungliga värden vara väldigt stor. En sidoeffekt av en större mängd möjliga ursprungliga värden kommer upp om destruktiv hashning används för att anonymisera primärntycklar och främmande nycklar; mängden av rader i databasen kan ändras. Detta kan vara

bra, eftersom det kan påverka nätverksutseendet. Om mängden noder ändras genom att olika noder och deras kanter läggs ihop kommer den anonymiserade grafen se annorlunda ut, och det kan vara svårare att identifiera grafen.

Även om ADS-GAN-metoden är intressant är den növändigtvis inte en praktiskt användbar lösning. Problemet med ADS-GAN är att det inte finns något som säger att den kan användas i en relationell databas. Däremot finns det källor [19] där motsatsen nämns. ADS-GAN har många fördelar, men den passar bättre för en mer enkel databasdesign. Det positiva med ADS-GAN är att den i teorin är bra för att skapa syntetiska data som liknar de ursprungliga datan, men inte är exakt lika eller för nära de ursprungliga datan.

10.7 Implementering

Till implementeringen bestämdes det att begränsa anonymiseringsprojektet till att bara använda två tabeller av datamodellen. Datamodellen är en relationsdatabas och alla tabeller har referenser till varandra eller är refererade av en annan tabell. De två tabellerna som fokuseras på kallas för Tab-1 och Tab-2. Tab-1 består av bland annat en primärnyckel (PK) som är en sträng av längden max 10 och är mänskligt läsbar. Tab-2 består av bland annat en primärnyckel som är ett 32bit positivt heltal och en främmande nyckel (FK) som refererar till primärnyckeln i Tab-1. Övriga intressanta data som de två tabellerna innehåller är datum.

För att utföra anonymisering av nätverksfaktorer (Som nämndes i sektion 10.3 [12]) utförs "binning", eftersom nätverken av anonymiserade rader inte bildar liknande mönster. Binning i detta fall implementeras med Faker-js [20] för PK i Tab-1 och med Seedrandom [23] för PK i Tab-2. För att Tab-2s FK ska vara synkroniserad med PK i Tab-1 måste operationen vara upprepbar. Det väljs att anropa *faker.seed* med existerande PK i Tab-1 eller FK i Tab-2 för att ge Faker ett frö. Efter att värdet är seedat kan till exempel *faker.company.name* anropas för att få ett binnat värde för det ursprungliga värdet. Binningsfaktorn testades för olika mängder syntetiska (inkrementerande siffror) och mera realistiska värden (skapad av o-seedad *faker.company.name*). Resultaten kan ses i Tabell2 och Tabell5.

Metoden som nämns ovan producerar mänskligt läsbara strängar längre än 10 tecken. Primärnykelfältet i Tab-1 kräver en max 10 tecken lång sträng. Strängen kan förkortas genom att förkorta vanliga ord som 'and' till '&' (Se Tabell 4) eller genom att ta bort vokaler (Se Kod 5). Med förkortningsmetoden erhålls närapå mänskligt läsbar text vars längd kan bestämmas genom att ange ett maximalt värde på inparametern.

För att göra binning för sifferbaserade fält som Tab-2s PK kunde *faker.numbers-.integer* användas, men det finns snabbare lösningar. Seedrandom [23] användes för att generera en seedad slumpmässig siffra med en viss decimalprecision. Med detta kan binning skapas snabbt (Se Kod 4 och Tabell 3)

Övrigt intressant som implementerades för att gynna anonymisering var tillägg av brus i kontinuerliga data som till exempel datumfält. Med datumfält används PK i Tab-1 och FK i Tab-2 för att seeda ett slumpmässig brusvärde vilket multipliceras med ett värde för att få en maximal brusnivå. PK i Tab-1 och FK i Tab-2 används för att ha samma relativa brusmängd för att undvika ordningsförändringar i datumfält mellan Tab-1 och Tab-2.

Fritextfält anonymiseras genom att räkna antalet ord och multiplicera strängen "words " med det antalet varefter vi tar bort sista mellanslaget. Övriga fält anonymiseras med binningmetoden men använder andra Faker-moduler.

10.8 Diskussion

Vissa problem med prestanda dök upp under implementeringen. I den skala som önskas presterar den versionen av Faker [20] som används inte tillräckligt bra (Se Tabell 6). För PK i Tab-1 och FK i Tab-2 måste en enklare binningsmetod anropas t.ex. genom att använda hashingalgoritmer. Metoden *faker.seed* i Faker är långsam att anropa. På grund av detta är de övriga fälten som anonymiseras med Faker-metoder inte seedade. Det innebär att om anonymisering körs flera gånger i rad på samma data, kommer resultaten att vara olika för de övriga fälten. Om prestanda inte är kritiskt kan seedade Faker-metoder användas för mer konsekventa anonymiseringar när det behövs. Man borde dock var försiktig med de frövärden som används,

eftersom fel frö kan avslöja data.

10.9 Slutsats

Denna avhandling visar att implementeringen är ett fördelaktigt alternativ till anonymisering av produktionsdata till bruk i testomgivningar. Genom att använda denna implementering kan en hög kvalitet på kod upprätthållas och testmetodik kan förbättras. Detta är viktigt för att säkerställa att systemen fungerar korrekt och pålitligt i produktionsmiljön, samtidigt som användning av riktiga produktionsdata för testbruk undviks.

References

[1] M. Bergeat, N. Cuppens-Bouhlahia, F. Cuppens, N. Jess, F. Dupont, S. Oulmakhzoune, and G. De Peretti, "A French anonymization experiment with health data," in *PSD 2014: Privacy in Statistical Databases*, 2014.

[2] R. Kumar, J. Novak, B. Pang, and A. Tomkins, "On anonymizing query logs via token-based hashing," in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 629–638.

[3] C.-C. Chou, "My three-day encounter with Argus: A report for iassist," *IASSIST Quarterly*, vol. 28, no. 4, pp. 8–8, 2005.

[4] J. Yoon, L. N. Drumright, and M. Van Der Schaar, "Anonymization through data synthesis using generative adversarial networks (ads-gan)," *IEEE journal of biomedical and health informatics*, vol. 24, no. 8, pp. 2378–2388, 2020.

[5] H. Li, L. Yu, and W. He, "The impact of gdpr on global technology development," pp. 1–6, 2019.

[6] M. Goddard, "The EU general data protection regulation (gdpr): European regulation that has a global impact," *International Journal of Market Research*, vol. 59, no. 6, pp. 703–705, 2017.

[7] P. Regulation, "General data protection regulation," *Intouch*, vol. 25, pp. 1–5, 2018.

[8] R. Motwani and S. U. Nabar, "Anonymizing unstructured data," *arXiv preprint arXiv:0810.5582*, 2008.

[9] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 111–125.

[10] Y. Lu, H. Wang, and W. Wei, "Machine learning for synthetic data generation: a review," *arXiv preprint arXiv:2302.04062*, 2023.

[11] R. N. Cardinal, "Clinical records anonymisation and text extraction (crate): an open-source software system," *BMC medical informatics and decision making*, vol. 17, pp. 1–12, 2017.

[12] B. Ouafae, R. Mariam, L. Oumaima, and L. Abdelouahid, "Data anonymization in social networks state of the art, exposure of shortcomings and discussion of new innovations," in *2020 1st International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*. IEEE, 2020, pp. 1–10.

[13] Facebook, "Global daily active users 2022," https://www.statista.com/statistics/346167/facebook-global-dau/, accessed: 2023-3-1.

[14] P.-P. de Wolf, "Java interface of mu-argus," https://github.com/sdcTools/muargus, n.d., accessed: May 17, 2023.

[15] M. Marx, E. Zimmer, T. Mueller, M. Blochberger, and H. Federrath, "Hashing of personally identifiable information is not sufficient," *SICHERHEIT 2018*, 2018.

[16] S. James, C. Harbron, J. Branson, and M. Sundler, "Synthetic data use: exploring use cases to optimise data utility," *Discover Artificial Intelligence*, vol. 1, no. 1, p. 15, 2021.

[17] D. J. Bernstein, "Quantum attacks against blue midnight wish, echo, fugue, grøstl, hamsi, jh, keccak, shabal, shavite-3, simd, and skein," 2010.

[18] J. Ali and V. Dyo, "Practical hash-based anonymity for MAC addresses," in *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications*. SCITEPRESS - Science and Technology Publications, 2020. [Online]. Available: https://doi.org/10.5220%2F0009825105720579

[19] L. Xu and K. Veeramachaneni, "Synthesizing tabular data using generative adversarial networks," *arXiv preprint arXiv:1811.11264*, 2018.

[20] Faker-Js, "Generate massive amounts of fake data in the browser and node.js." [Online]. Available: https://github.com/faker-js/faker

[21] [Online]. Available: https://www.getpostalcodes.com/vatican/place-citta-del-vaticano/

[22] G. Sparks, "Database modelling in uml," *Methods & Tools*, vol. 9, no. 1, pp. 10–23, 2001.

[23] D. Bau, "seeded random number generator for javascript." [Online]. Available: https://github.com/davidbau/seedrandom

[24] R. Roshdy, M. Fouad, and M. Aboul-Dahab, "Design and implementation a new security hash algorithm based on md5 and sha-256," *International Journal of Engineering Sciences & Emerging Technologies*, vol. 6, no. 1, pp. 29–36, 2013.

[25] G. Sampson, "Writing systems," *London, UK: H utchinson*, 1985.

[26] A. P. Cibois, "Lorem ipsum: Nouvel état de la question," Mar 2012. [Online]. Available: https://idm.hypotheses.org/2354