Technische Universität Berlin

Chair of Database Systems and Information Management

Master's Thesis

Assessing Privacy of Disclosure-Compliant Policies in Databases

Irild Hoxhallari
Degree Program: Computer Science
Matriculation Number: 487673

Reviewers
Prof. Dr. Volker Markl
Prof. Dr. Matthias Böhm

Advisor(s)
Rudi Poepsel Lemaitre

Submission Date
November 30, 2025

I hereby declare that the thesis submitted is my own, unaided work, completed without any external help. Only the sources and resources listed were used. All passages taken from the sources and aids used, either unchanged or paraphrased, have been marked as such. Where generative AI tools were used, I have indicated the product name, manufacturer, the software version used, as well as the respective purpose (e.g. checking and improving language in the texts, systematic research). I am fully responsible for the selection, adoption, and all results of the AI-generated output I use. I have taken note of the Principles for Ensuring Good Research Practice at TU Berlin dated 15 February 2023.

I further declare that I have not submitted the thesis in the same or similar form to any other examination authority.

Berlin, November 30, 2025

.....

Firstname, Lastname(s)

Acknowledgments

I want to express my sincere gratitude to my advisor Rudi Poepsel Lemaitre for his constant support and encouragement during this project. His ideas, comments and suggestions were very helpful and made achieving the goals and objectives of this thesis possible. His expertise and insights greatly shaped the direction and quality of this work.

I also want to thank Prof. Dr. Volker Markl and the Chair of Database Systems and Information Management as a whole for making the possibility of working on such an interesting topic a reality and helping me become a better researcher and professional in the field of computer science. Furthermore, I would like to thank all the professors and tutors who dedicate their time to shaping the minds of the future and giving guidance to all students, including me, in becoming the best version of themselves academically and professionally.

I especially want to thank my family, for their unconditional love and support during my studies and giving me the confidence and strength to continue working on and bettering myself in all facets of my life. I want to specifically thank my parents for their sacrifices that made studying in a foreign country far away from home possible. I will always appreciate how hard you work to enable me to achieve my dreams. I also want to thank my brother Erdi for his company and constant motivation and support during my toughest moments and for always being there for me when I needed him most. Without your constant encouragement and strong belief in me, I could have never achieved what I have until now.

Lastly, I want to mention and thank my wife Tomorjana, for her love and support during my studies. You endured the sleepless nights and stressful days along with me and made everything possible by believing in me and making me think I am the most capable man in the world. You brought color to my life and made everything better in it. Thank you. Të dua.

Zusammenfassung

Moderne datenintensive Systeme stehen vor zunehmenden und fortlaufenden Herausforderungen, die Einhaltung von Datenschutzvorschriften wie der GDPR zu erreichen und aufrechtzuerhalten, insbesondere beim Umgang mit dynamischen Datensätzen. Während Systeme wie Mascara Offenlegungsrichtlinien während der Anfrageausführung durchsetzen können, fehlen ihnen Mechanismen, um sicherzustellen, dass diese Richtlinien im Laufe der Zeit wirksam bleiben oder gegenüber Schwachstellen robust sind, die durch sich verändernde, dynamische Daten entstehen. Diese Arbeit befasst sich mit dem Problem der Bewertung und Aufrechterhaltung der Datenschutzgarantien bei richtlinienkonformen Offenlegungsanfragen in relationalen Datenbanken mit kontinuierlich veränderten Daten.

Zur Lösung dieses Problems entwerfen und implementieren wir *PolicyLiner*, ein Datenbank-Auditing-Tool, das Online- und Offline-Auditing kombiniert, um Offenlegungsrichtlinien und Offenlegungsanfragen kontinuierlich zu überwachen und zu evaluieren. Der Online-Auditing-Prozess führt eine Echtzeitanalyse von Anfragen durch, um verdächtige Muster zu erkennen, die zu Schwachstellen gegenüber *Query Replay Attacks* führen können, während der Offline-Auditing-Prozess historische Anfragen retrospektiv untersucht und Offenlegungsrichtlinien anhand verschiedener Datenschutzmetriken bewertet, darunter t-Closeness, δ-Presence, Sample Uniqueness Ratio und Population Uniqueness Estimation. Durch eine umfassende experimentelle Evaluation zeigen wir, dass *PolicyLiner* mehrere Arten von Privacy-Angriffen effektiv identifiziert, einen geringen Anfrage-Overhead aufweist und unter gleichzeitigen Workloads skaliert. Unsere Ergebnisse zeigen, dass *PolicyLiner* Schwachstellen gegenüber verschiedenen Privacy-Angriffen wie *Membership Disclosure Attacks*, *Attribute Disclosure Attacks*, *Re-identification Attacks* und *Query Replay Attacks* aufdecken kann. Damit erhält der Datenschutzbeauftragte die nötigen Werkzeuge und die erforderliche Weitsicht, um diese Angriffe rechtzeitig zu verhindern und die langfristige Robustheit sensibler Datenbanksysteme erheblich zu stärken, indem sichergestellt wird, dass richtlinienkonforme Offenlegungsrichtlinien korrekt und aktuell bleiben.

Abstract

Modern data-intensive systems face growing, ongoing challenges in achieving and maintaining compliance with privacy regulations such as GDPR, especially when dealing with dynamic datasets. While systems like Mascara can enforce disclosure policies during query execution, they lack mechanisms to ensure these policies remain effective over time or resilient against vulnerabilities introduced by evolving, dynamic data. This thesis addresses the problem of assessing and preserving the privacy guarantees of disclosure-compliant policies and queries in relational database environments with continuously changing data.

To solve this, we design and implement PolicyLiner, a database auditing tool that combines online and offline auditing to monitor and continuously evaluate disclosure policies and disclosure queries. The online auditing process performs real-time query analysis to detect suspicious patterns that can lead to vulnerabilities towards query replay attacks, while the offline auditing process retrospectively examines historical queries and evaluates disclosure policies using various privacy metrics, including t-Closeness, δ-Presence, Sample Uniqueness Ratio, and Population Uniqueness Estimation. Through a comprehensive experimental evaluation, we demonstrate that PolicyLiner effectively identifies multiple types of privacy attacks, maintains low query execution overhead, and scales under concurrent workloads. Our results show that PolicyLiner can discover vulnerabilities to various privacy attacks, such as *Membership Disclosure Attacks*, *Attribute Disclosure Attacks*, *Re-identification Attacks*, and *Query Replay Attacks*, giving the data officer the needed tools and foresight to prevent these attacks on time and significantly strengthen the long-term robustness of sensitive database systems by ensuring that disclosure-compliant policies remain accurate and up-to-date.

Contents

1 Introduction

In recent years, data privacy and the protection of personal data have become one of the central challenges in information systems and database management. With the widespread, large-scale data collection and processing that occur today, organizations are under increasing pressure to ensure compliance with privacy regulations such as the General Data Protection Regulation (GDPR) in the European Union[13]. Beyond regulatory obligations, maintaining trust with users requires that sensitive information be handled with care, particularly in environments where sensitive data are stored and accessed on a regular basis.

That is why organizations are continuously looking for easier and more robust ways to implement data privacy and protect the personal data of their customers. Disclosure-compliant policies, enforced by data officers and middleware tools such as Mascara [38], are designed to shield sensitive information from unauthorized access. However, they might not be enough in databases with rapidly evolving and dynamic data. Once-effective disclosure policies can be unintentionally rendered obsolete by the ever-changing data, possibly causing privacy breaches and regulatory violations if left unchecked.

1.1 Problem Statement

While existing database auditing solutions primarily focus on data integrity and consistency [18, 16, 23], there is a significant gap concerning the ongoing assessment and supervision of disclosure-compliant policies in environments with dynamic data. Current query auditing tools [26, 24, 19] lack mechanisms to continuously monitor, evaluate, and update disclosure policies in response to data alterations and while taking into account user query histories, thereby risking non-compliance and exposure of sensitive information.

In this work, we aim to design and create a database auditing tool called *PolicyLiner* that can implement mechanisms to monitor, evaluate, and keep disclosure policies up-to-date. We want to investigate if this can be done in an efficient, accurate, and easy-to-use way. In order to achieve this, we will be using two ways of auditing: online auditing and offline auditing. We define online auditing as the process of continuously monitoring database operations and queries in real time, allowing for immediate detection and response to any anomalies or potential

privacy attacks. Offline auditing is defined as the periodic and retrospective examination of previous disclosure queries and current disclosure policies stored in the audit database to assess compliance and detect issues that may have occurred over a certain past period.

To measure efficiency, we want to explore the overhead our auditing brings to the execution of user queries and assess if that delay is significant or negligible. Additionally, we want to measure the robustness of our online auditing process by simulating concurrent users in a controlled environment. To measure accuracy, we inspect different kinds of use cases, where different privacy attacks might occur, and examine whether PolicyLiner can catch and prevent these attacks from happening. To make our tool as easy to use as possible, we create a standalone application with a REST API to call all the functions of PolicyLiner. In this way, making PolicyLiner usable by anyone with just an API call.

1.2 Contributions

In this thesis, we present the results of our work and highlight the advantages of using our database auditing tool when dealing with disclosure queries and policies. In summary, we make the following significant contributions:

Literature and Research Review: In Chapter 2, we introduce the core findings of our literature research regarding database auditing and various privacy attacks and metrics.

Online and Offline Analysis: In Chapter 3, we describe our approach to auditing disclosure queries and policies. We separate the different functions into online and offline analysis, depending on the overhead they add to the process, and use non-blocking separate virtual threads to ensure the best performance possible.

PolicyLiner Implementation: We introduce PolicyLiner, our database auditing tool that implements the analysis functions devised in chapter 3 in connection to a state-of-the-art system for disclosure-compliant query answering called Mascara[38]. We also describe how this tool was created and its main components and features (Chapter 4).

Evaluation: We experimentally evaluate PolicyLiner and its auditing functions in terms of effectiveness and overhead, constructing experiments for each of the functions over a handpicked dataset (Chapter 5). We evaluate PolicyLiner's robustness and scalability to heavy concurrent query evaluations by multiple users.

With our results, we demonstrate the advantages of using PolicyLiner, showing that our tool can recognize different kinds of privacy attacks and report them before they can leak any data to unwanted third parties.

2 Literature and Research Review

This chapter introduces the findings of our literature research on existing database auditing systems and describes the theoretical foundations of the database auditing tool that we have created. We first present previous auditing systems, what they do, and how they achieve their goals. Then, we describe the concepts of Anonymization and Data Masking. Based on this, we introduce Mascara, the system whose functionality our tool extends. Lastly, we detail different database privacy attacks and metrics that are relevant to our thesis.

2.1 Database Auditing

Hiremath et al. [18] present a public auditing technique for cloud computing that aims to ensure data privacy and integrity using the Third Party Auditor (TPA). The proposed auditing scheme uses the AES algorithm for encryption and the Secure Hash Algorithm (SHA-2) to generate verification metadata, or message digests, for data integrity checks [18]. Compared to prior works, which often address either privacy or integrity but struggle to maintain both simultaneously, this approach achieves constant-time auditing across various file sizes and ensures secure verification without disclosing sensitive information to the Third Party Auditor.

Groomer et al. [16] introduce an approach for continuous auditing in database environments using embedded audit modules (EAMs), which are sections of code incorporated into application programs to capture audit-related information as transactions occur. The EAMs provide auditors automated and ongoing access to data about control and access violations, supporting both compliance and substantive testing on a continuous basis. Unlike traditional periodic auditing or approaches reliant only on database management system security features, this method addresses unique control and security concerns in database applications by storing violation data in dedicated tables for auditors to review. This comes at the cost of added complexity and requires specialized technical knowledge from the auditors.

Liu et al. [23] present a framework of database auditing, which log the database activities through analyzing network traffic, execute audit analysis through event correlation and generate alarms if an anomaly or a violation of security regulations

is detected. They claim that their approach has the advantage of providing zero impact on the performance of the database or the applications that access it[23]. In addition, their approach provides increased reliability, and it adheres to the separation of duties by storing audit logs on an independent server. However, as they use network-based logging, if the database communication has been encrypted, their approach becomes invalid.

Zeng et al. [54] introduce an automated approach to abstracting behaviors by inferring and aggregating the semantics of audit events called WATSON. WATSON leverages a translation-based embedding model to infer the semantics of audit events based on contextual information in logs. It summarizes relevant behaviors, groups semantically similar patterns together, and picks out a representative example for analyst review. **Zeng et al.** evaluate WATSON against both behaviors simulated from real-life cyber attacks as well as behaviors of an adversarial engagement. Their experimental results show that WATSON can accurately abstract both benign and malicious behaviors and dramatically reduce manual workload in attack investigation [54].

Anwar et al. [4] present a database auditing method based on synchronization and row-based auditing, where each data change is logged as a new record in a dedicated audit table, separated from the operational database. This design allows database administrators to efficiently track historical data changes in real time and improves the speed and reliability of audit queries by decoupling audit workloads from transactional workloads. Their approach supports a detailed review of data operations and can help detect anomalies, although it does not capture all database actions, such as DDL and DCL events, which cannot be logged.

As we can see, all the aforementioned papers address database auditing in one capacity or another. Some audit the data integrity in databases, others audit data security and anomalies that shouldn't happen to the data in a financial or otherwise sensitive setting. While these use cases do not align with our database auditing goals, which aim to ensure the privacy of disclosure queries and policies in databases, we can still learn from the methods described in these papers to improve our work.

2.2 Query Auditing

Liu et al. [24] focus on the privacy protection and integrity audit methods of tenant outsourced data under cloud environments. Through the in-depth analysis of the current research on privacy protection and data integrity audit, the existing security threats are found, and a series of solutions are proposed, including the privacy protection scheme based on data block confusion and data coloring, and the multi-copy data integrity audit method supporting dynamic data update.

Unlike older methods, such as simple encryption or K-anonymity, this framework can better trace and protect large datasets, while allowing for real-time audits through efficient cryptographic protocols like homomorphic signatures and structured tree authentication. After experimental verification and theoretical analysis, these methods achieve great results in maintaining the accuracy, availability, and integrity of tenant cloud data [24].

Motwani et al. [26] study the problem of auditing a batch of SQL queries with respect to "forbidden views" (sets of confidential data that should remain private). A significant contribution of this paper is the introduction and definition of different notions of suspiciousness, especially "weak syntactic suspiciousness", that are independent of the database instance, with the hope that by eliminating dependence on underlying data, suspiciousness can be determined from just the structure of the queries. The weak syntactic auditor in conjunction with a set of association views can be used to detect queries that potentially violate the association privacy of an individual and his sensitive attributes (An attribute is marked sensitive if an adversary must not be allowed to discover the value of that attribute for any individual in a dataset[25]).

Ingle et al. [19] provide a comprehensive overview of database query auditing techniques. They focus on a framework for auditing queries and several different notions of suspiciousness for simple SQL queries. The paper also focuses on the notion of SELECT triggers that extend triggers to work for SELECT queries in order to facilitate data auditing. The authors also discuss the concept of batch query auditing to detect indirect disclosure of confidential data. They conclude by highlighting the benefits of SELECT triggers, such as the possibility of real-time feedback on access to sensitive data and the fact that they reduce the overall auditing runtime by filtering queries and their associated accesses that must be analyzed by an offline system [19].

We see in these papers different ways of auditing database queries in a cloud environment and in a SQL database setting. While their contributions are significant and valuable, they both deal with different use cases and address a broader spectrum of queries, unlike our work, which concentrates on the auditing of disclosure-compliant queries. Another shortcoming of these papers is that they do not audit the privacy policies that define the needed privacy protection in the database.

2.3 Privacy Auditing

Prasser et al. [40] present ARX, an open-source anonymization tool, that implements a wide variety of privacy methods in order to make the distribution of data safer and more robust against privacy threats, especially in the biomedical research field[40]. ARX offers a user interface and a programming API, so that

users or developers can work with it and develop it further.

The limitation of this work comes from the fact that they use query-agnostic data masking to anonymize the data regardless of the query issued by the user [38]. They also have no mechanism for checking the continuous privacy of a constantly changing dataset. In our thesis, we address both of these concerns. Mascara [38] is used as the data masking software for anonymizing the user queries into disclosure-compliant ones, and PolicyLiner is added to it to address the constantly changing attributes of datasets, so that the initially defined disclosure policies are kept up-to-date, and the privacy of users is maintained.

2.4 Data Anonymization

Anonymization is the process of making personal data anonymous [47]. According to the European Union's data protection laws, in particular the General Data Protection Regulation (GDPR), anonymous data is "information which does not relate to an identified or identifiable natural person or to personal data rendered anonymous in such a manner that the data subject is not or no longer identifiable"[47]. There are many ways to anonymize data.

Pseudonymization replaces direct identifiers (such as names) with pseudonyms or fake identifiers to prevent direct identification of individuals. Unlike complete anonymization, pseudonymized data can be re-identified using a key that links the pseudonyms to real identities [50].

Generalization is a process of replacing the original value with a less specific but semantically consistent value. This technique applies to the cell level, where some original values are maintained with additional confusion [27].

Suppression refers to the removal of an entire part of data (column or tuple) in a dataset by changing the value to one value that doesn't have meaning (eg, "****"), replacing the original data [27].

Perturbation adds random noise or otherwise distorts data to prevent the inference of sensitive information while keeping aggregate statistics useful [2].

Data Masking is a method of creating a structurally similar but inauthentic version of an organization's data[1]. The purpose is to protect the actual data while having a functional substitute for occasions when the real data is not required [1]. Data masking keeps sensitive information private by making it unrecognizable but still usable. This allows developers, researchers, and analysts to use a data set without exposing the data to any risk [1].

2.5 Mascara: Access Control with Data Masking

Our thesis is built upon the paper and work of Rudi Poepsel Lemaitre et al. titled "Disclosure-Compliant Query Answering" [38]. In this work, they develop a middleware called Mascara with the aim of specifying and enforcing data disclosure policies. They introduce data masks to specify disclosure policies more flexibly and intuitively and propose a query modification approach to rewrite user queries into disclosure-compliant ones [38]. In Figure 1 we see an illustration of the schematics

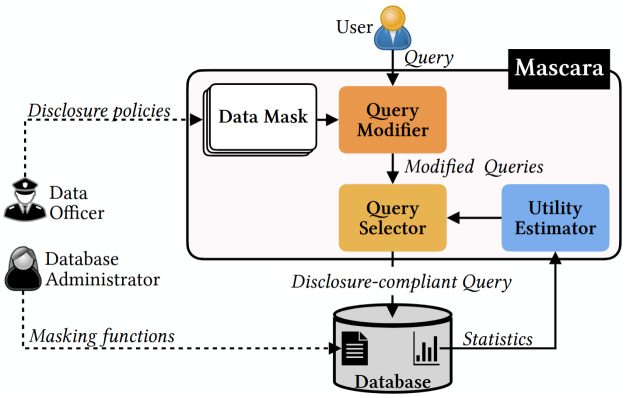

Figure 1: MASCARA Overview[38].

of Mascara. A **data mask** specifies which information attributes can be disclosed to whom and how. Each masking function is implemented as a User-Defined Function (UDF) that is registered within the database system. During query execution, Mascara validates each query to determine whether it accesses restricted data. It then modifies the queries by parsing the query into its logical plan and then modifies each plan operator so that the query respects the disclosure policy. If the **query modifier** fails to rewrite a query, it rejects it[38]. In cases where multiple policies apply, Mascara generates several disclosure-compliant query alternatives and uses the **query selector** to choose the version that best balances disclosure policy compliance with minimal information loss. This selection is based on **utility estimation**, which measures expected data quality using database statistics and masking function properties[38].

2.6 Database Privacy Attacks

In this section, we describe some of the most common privacy attacks used by malicious third parties to get access to private personal data.

2.6.1 Linking Attack

A linking attack is an approach for exposing information specific to individuals in a de-identified dataset by matching up records with a second dataset [29].

2.6.2 Homogeneity Attack

A homogeneity attack occurs when all records within an equivalence class share the same value for a certain sensitive attribute. Even though individuals cannot be distinguished based on quasi-identifiers, the uniformity of the sensitive attribute allows an adversary to infer that sensitive value with certainty[25].

Quasi-identifiers are attributes whose values, when taken together, can potentially identify an individual[22]. **Equivalence Classes** are defined as a set of records that have the same values for the quasi-identifiers[22].

2.6.3 Background Knowledge Attack

A background knowledge attack exploits the adversary's external or prior knowledge to narrow down or infer the true sensitive value of an individual, even when multiple possible sensitive values exist within an equivalence class[25].

2.6.4 Membership Disclosure Attack

Membership disclosure means that linkage allows to determine whether or not data about an individual is contained in a dataset[30]. While this does not directly disclose any information from the dataset itself, it may allow an attacker to infer meta-information. If, for example, the data is from a cancer registry, it can be inferred that an individual has or has had cancer[39].

2.6.5 Attribute Disclosure Attack

Attribute disclosure means that an attacker can infer sensitive attribute values about an individual, i.e., information with which individuals are not willing to be linked, without necessarily relating the individual to a specific record in a dataset [39]. As an example, linkage to a set of records can allow inferring information if all records share a certain sensitive attribute value[39].

2.6.6 Re-identification Attack

Identity disclosure (or re-identification) means that an individual is linked to a specific data record[39]. This type of attack is addressed by many laws and regulations worldwide, and it is therefore often related to severe consequences for data

owners. From the definition of this type of disclosure, it follows that an attacker can learn all sensitive information contained about the individual [39].

2.6.7 Query Replay Attack

An adversary causes or observes the same or similar queries being issued multiple times to refine knowledge about which items are being queried/leaked. By repeating queries and observing results (volumes, leakage, response times), the adversary can gradually reduce uncertainty [37].

2.7 Privacy Metrics

In this section, we would like to introduce some relevant privacy metrics that can be used to evaluate the privacy of any certain disclosure query or policy.

2.7.1 k-Anonymity

Sweeney et al. [48] define a dataset as k-Anonymity protected if the information for each person contained in the dataset cannot be distinguished from at least $k-1$ individuals whose information also appears in the release. This metric is important because it forms the basis on which many other metrics and privacy-protecting systems are built[48].

2.7.2 l-Diversity

A k-anonymized dataset has some privacy problems. First, an attacker can discover the values of sensitive attributes when there is little diversity in those sensitive attributes. Second, attackers often have background knowledge, and k-anonymity does not guarantee privacy against attackers using background knowledge [25]. That is why Machanavajjhala et al. [25] proposed a new metric called l-diversity, which solves the above-mentioned problems. L-diversity extends k-anonymity, requiring that each equivalence class must be associated with at least l well-represented values for a sensitive attribute. Machanavajjhala et al. propose different definitions for "well-represented" values [51]. The simplest formalization of l-diversity requires that each equivalence class include at least l different values for the sensitive attribute[51].

2.7.3 t-Closeness

According to Li et al., l-diversity is neither necessary nor sufficient to prevent attribute disclosure[22]. One problem with l-diversity is that it is limited in its

assumption of adversarial knowledge. It is possible for an adversary to gain information about a sensitive attribute as long as she has information about the global distribution of this attribute. This assumption generalizes the specific background and homogeneity attacks used to motivate l-diversity. Another problem with l-diversity and privacy-preserving methods in general is that they effectively assume all attributes to be categorical; the adversary either does or does not learn something sensitive. Of course, especially with numerical attributes, being close to the value is often good enough [22].

Li et al. [22] proposed another privacy-preserving method called t-closeness. They define their t-closeness principle as:

An equivalence class is said to have t-closeness if the distance between the distribution of a sensitive attribute in this class and the distribution of the attribute in the whole table is no more than a threshold t. A table is said to have t-closeness if all equivalence classes have t-closeness[22].

2.7.4 δ-Presence

The δ-Presence metric was proposed by Nergiz et al. [30] as a metric to evaluate the risk that a data recipient can identify the presence of an individual in a dataset [51]. This means that it was created with the aim of countering membership inference attacks [39]. The mathematical definition given by Nergiz et al.[30] for this metric is:

Given an external public table P, and a private table T, we say that δ-presence holds for a generalization T^* of T, with $\delta = (\delta_{\min}, \delta_{\max})$, if

$$\delta_{\min} \leq P(t \in T \mid T^*) \leq \delta_{\max} \qquad \forall\, t \in P \tag{2.1}$$

In such a dataset, we say that each tuple $t \in P$ is δ-present *in T. Therefore, $\delta = (\delta_{\min}, \delta_{\max})$ is a range of acceptable probabilities for $P(t \in T \mid T^*)$.*

Tuning the $\delta_{\min}$ and $\delta_{\max}$ values allows finding the optimal trade-off between data utility and privacy for a given dataset. A small $[\delta_{\min}, \delta_{\max}]$ range favors privacy, while a large $[\delta_{\min}, \delta_{\max}]$ range favors data utility[51].

2.7.5 Uniqueness Estimation

Uniqueness, which is defined as a unique record being in an equivalence class of size one [8], is also a metric for measuring re-identification risk of an individual [3]. There are two types of uniqueness metrics: *Sample uniqueness* and *Population uniqueness*.

Sample uniqueness measures the proportion of records that are unique with respect to a set of quasi-identifiers within the released dataset [8]. Population uniqueness refers to a situation where an individual's combination of quasi-identifiers

occurs only once in the entire target population. Since population data are rarely available, population uniqueness is generally estimated from the observed sample uniqueness using probabilistic or modeling techniques [45].

Population uniqueness is a commonly utilized measure of re-identification risk [3]. Unique records are more likely to be re-identified than non-unique records [3]. Uniqueness can be measured in several ways. Mainly, it is the probability of a record being unique in the original dataset [3]. Another approach would be measuring the proportion of records in a public dataset that are unique [8]. The following equation measures the proportion of records in a given dataset that are unique[8]:

$$\lambda = \frac{\sum_i I(F_i = 1)}{N} \tag{2.2}$$

I is an indicator function. $I(F_i = 1)$ is 1 if the record is unique in the corresponding equivalence class, and 0 otherwise. Having said that, it is not possible to measure uniqueness precisely, therefore it should be estimated[3].

Various ways have been proposed in previous literature to estimate the uniqueness of a sample dataset. Dankar et al. [8] conducted evaluations of different uniqueness estimators and applied Pitman[36], Zayatz[53], and SNB[7] uniqueness estimators on a clinical dataset and concluded that there was no single estimator that performed well across all conditions. Pitman estimator was the most accurate of all, but only for low sampling fractions, while the SNB and Zayatz performed equally accurately for the higher sampling fraction [3]. Hence, a decision rule shown in Algorithm 1 was adopted by Dankar et al.[8], which chooses appropriate models based on sampling fraction.

Algorithm 1 Decision Rule of Dankar et al.[8] for choosing the appropriate model(E_1)

```
if π ≤ 0.1 then
    E1 ← Pitman
else
    if SNB converges then
        if Est(SNB) > Est(Zayatz) then
            E1 ← Zayatz
        else
            E1 ← SNB
        end if
    else
        E1 ← Zayatz
    end if
end if
```

3 Approach: Database Privacy Auditing

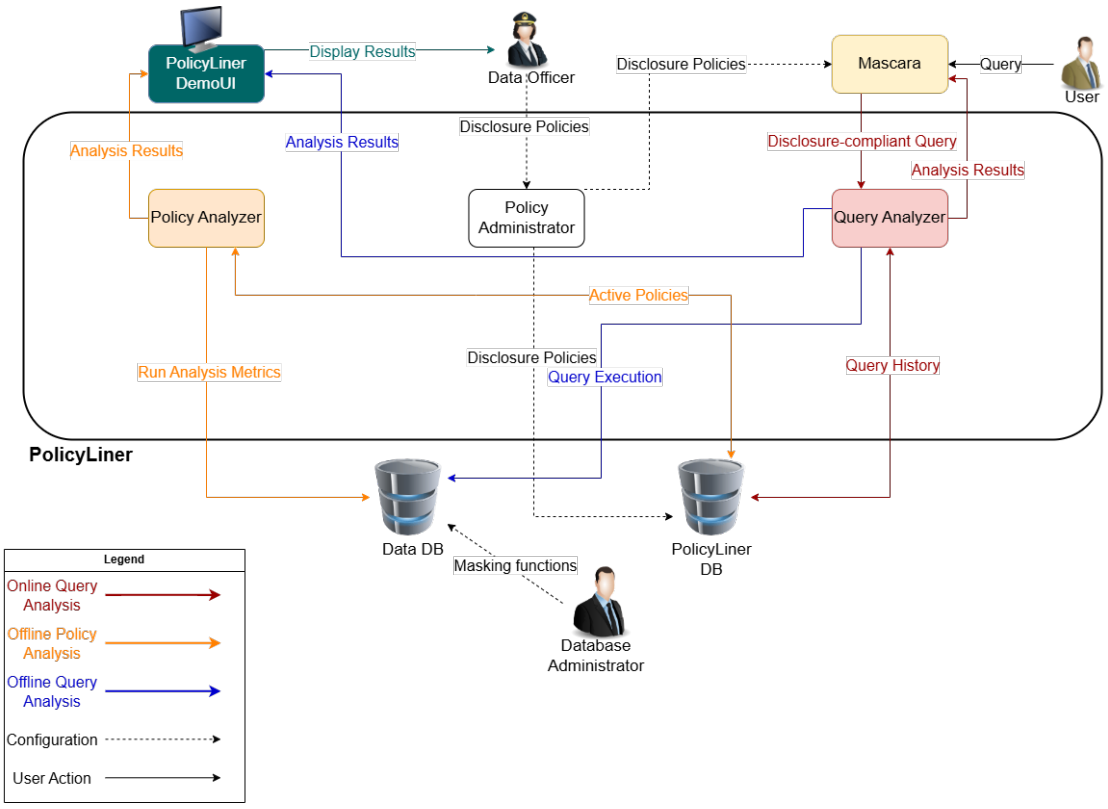

Figure 2: PolicyLiner Architecture

After exploring and describing the necessary background knowledge in the previous chapter, we can start to explain our approach to database privacy auditing. In this work, we propose using two different types of auditing: online auditing and offline auditing. With these two types of auditing, we aim to discover in a timely manner and ideally prevent four different types of privacy attacks: Query Replay Attack, Membership Disclosure Attack, Attribute Disclosure Attack, and Re-identification Attack. Additionally, we create a policy administrator compo-

nent, which can parse policies from the Mascara *data mask* DDL-like statements and activate them in the database.

In Figure 2, we depict the PolicyLiner system and workflows between its different components. At the core of the PolicyLiner system lies a set of interconnected, modular components, each focused on a specific aspect of disclosure policy and query auditing.

Now we would like to describe how each module connects to the others and how everything flows. First, the Database Administrator defines the needed masking functions in the primary database, which contains the user data (called "Data DB" in figure 2). Then the Data Officer defines the disclosure policies of the database using Mascara *data mask* statements. These statements are parsed by the **Policy Administrator** component, the policies are processed, activated, and stored in the PolicyLiner Database. They are then also passed on to Mascara. After this action, the configuration process is complete, and the auditing processes can now be started.

There are two components responsible for auditing disclosure queries and policies and making sure that the privacy of the user data is maintained: **Policy Analyzer** and **Query Analyzer**.

Policy Analyzer operates almost exclusively in the *offline auditing* processes and is explicitly responsible for completing the *Offline Policy Analysis*. The process starts by pulling the active policies from the PolicyLiner database into the Policy Analyzer component. Then the previously defined *Analysis / Privacy metrics* are run on the real data in the primary database. The findings are gathered into one object and sent as *Analysis Results* to the **PolicyLiner DemoUI**, which then displays the findings to the data officer. In this way, the data officer can clearly see when a policy is no longer compliant or up to date and can take the appropriate action via the Policy Administrator.

Query Analyzer acts in both types of auditing, online and offline. It is responsible for the *Online Query Analysis* and *Offline Query Analysis* processes.

- The *Online Query Analysis* starts when a user wants to run a query on the primary database. This query is at first handled by Mascara and converted into a disclosure-compliant query. This disclosure query is then sent to the Query Analyzer for auditing. The Query Analyzer then pulls all previous queries from the same user and evaluates the user activity as a whole for suspicious actions in real time. The results of this analysis are gathered and sent to Mascara as the *Analysis Results* together with the decision if this query should be run, denied, or if this user should be observed more closely by the Data Officer.

- The *Offline Query Analysis*, in contrast, is run periodically independent of the user actions. The Query Analyzer first pulls the complete unaudited

query history from the PolicyLiner database. These queries are evaluated on a user-by-user basis and also executed as needed in the primary database in order for the query results to be accurately analyzed. These results are thoroughly evaluated and, similarly to the Online Query Analysis, are collected into the *Analysis Results* object. However, this object is now sent to the PolicyLiner DemoUI for the Data Officer to assess.

In the next sections of this chapter, we will be describing the aforementioned processes in depth and give an extensive explanation as to how PolicyLiner works under the hood.

3.1 Configuration

The creation and definition of disclosure policies is an important step, if not the most important step, in keeping the privacy of the user data intact. With our **Policy Administrator**, we aim to make this process as easy and seamless as possible.

Poepsel-Lemaitre et al. propose a new language to define disclosure policies called *data mask* that allows Data Officers and Database Administrators to conveniently specify disclosure policies [38]. The structure and syntax of a data mask for a particular disclosure policy P is shown in code fragment 1.

Code Fragment 1: Data Mask syntax [38]

```
disclose attribute list from table [, ...]
[with mask on attribute using masking_function] [, ..]
[where condition]
```

We follow this language of defining disclosure policies and create the functionality inside of Policy Administrator to parse, understand, and translate *data mask* into a language the database can understand (SQL). This allows the Data Officer to define disclosure policies in PolicyLiner and have them distributed to Mascara and stored in the PolicyLiner Database within the same workflow. An example of this functionality is shown in fragments 2 and 3, where fragment 2 is the code the Data Officer writes to define a disclosure policy and fragment 3 is the translated and database-understandable code from the Policy Administrator.

Code Fragment 2: Disclosure Policy Definition Example

```
disclose patients.patientid, patientlanguage,
   patientpopulationpercentagebelowpoverty,
            admissions.admissionid, primarydiagnosiscode
    from patients, admissions, diagnosis
    with mask on patients.patientid using suppress()
```

```
 with mask on admissions.admissionid using suppress()
 with mask on patientlanguage using
randomized_response_patientlanguage()
 with mask on primarydiagnosiscode using
generalize_primarydiagnosiscode()
 where patients.patientid = admissions.patientid
 and admissions.admissionid = diagnosis.admissionid
 and $user.role = 'public_health_researcher'
```

Code Fragment 3: Disclosure Policy SQL Translation Example

```
CREATE VIEW patients_admissions_diagnosis_policy
    AS SELECT
        suppress(patients.patientid) AS patients_patientid,
        suppress(admissions.admissionid) AS admissions_admissionid,
        randomized_response_patientlanguage(patientlanguage)
        AS patientlanguage,
        generalize_primarydiagnosiscode(primarydiagnosiscode)
        AS primarydiagnosiscode,
        patientpopulationpercentagebelowpoverty
        AS patientpopulationpercentagebelowpoverty
    FROM patients
        JOIN admissions
        ON admissions.patientid = patients.patientid
        JOIN diagnosis
        ON diagnosis.admissionid = admissions.admissionid;
```

In addition to creating this relational-database-readable code, we store more metadata about the disclosure policy in our PolicyLiner database. As we see in fragment 2, *data mask* gives us the option of defining the user role that has access to this particular disclosure policy. While we cannot translate this information into SQL, we can store it in our PolicyLiner database.

Code Fragment 4: Disclosure Policy Data Structure

```
1     {"disclosurePolicy": {
2             "policy": "string",
3             "allowedUserRole": "string",
4             "createdAt": "datetime",
5             "viewName": "string",
6             "status": "enum",
7         }
8     }
```

The data structure of the information stored about each disclosure policy in our database is shown in code fragment 4. The *Policy* attribute stores the translated disclosure policy creation statement. The *allowedUserRole* attribute stores the user role mentioned in the data mask statement. The *createdAt* attribute stores the time at which the disclosure policy was created. The *viewName* attribute stores the name of the disclosure policy. Lastly, the *status* attribute defines if the policy is active or deactivated. Changing the status of a disclosure policy is also part of the Policy Administrator's functionality. The policy is deactivated when it is no longer up-to-date or useful, according to the data officer.

3.2 Online Analysis

Query Replay Attacks are caused by the repetition of the same or similar queries to detect information leaks from the database. Adversaries can discover real sensitive information about the users of the system by replaying the same or similar queries when the implemented disclosure policies use dynamic masking functions that change the output in various ways with every query execution.

This is what we want to prevent with our Online Query Analysis workflow. We propose an implementation that detects when the same user is running numerous similar queries, and we act on the results of that analysis.

3.2.1 Online Query Auditing

The Online Query Analysis is started in PolicyLiner when we receive the disclosure-compliant query object, described in code fragment 5 from Mascara.

Code Fragment 5: Query Object Request

```
{
    "queryObject": {
        "query": "string",
        "userId": "string",
        "userRole": "string",
        "comparatorType": "enum"
    }
}
```

As we can see, in addition to the rewritten query by Mascara, this object contains other metadata that helps PolicyLiner in its analysis. First, we have the *userId*, whose value contains a unique identifier for every user who aims to run queries in the primary database. Then comes the *userRole* value, which contains the role that the user has in the system. Lastly, we have the *comparatorType*, which defines the

way that queries are compared to one another. There are three comparison types in our online query analysis: simple string comparison, Levenshtein distance, and our own custom implementation. The Levenshtein distance is a metric for measuring the difference between two string sequences. The Levenshtein distance between two strings is the minimum number of single-character edits required to change one word into the other [21]. In contrast to the two other options, our own custom implementation aims to compare the queries with one another, not just as strings, but as statements that do not need to be completely similar string sequences to get comparable results from the database.

After we receive the request from Mascara with the query object, we can start the evaluation of the requested query. The goal of this analysis is to prevent the aforementioned query replay attacks. We aim to achieve this goal by comparing all historical queries of one user with the current query and detecting if the current query request is similar to any of them.

The first step of the evaluation, as seen in Algorithm 2, is checking if the user exists in our database. If this user does not exist in our PolicyLiner database, it is then created, and the query is approved. The query is also immediately approved if the user already exists but has not run any previous queries. In the event that the user has already executed more than one other query, we start comparing them. We loop through all the previous queries and compare them with the current query. The comparison is done depending on the value of the *comparatorType* attribute, and all the queries deemed to be *similar* are collected into a list of similar queries. Depending on the length of this list, we determine the decision that needs to be taken about the current query. If there are no other similar queries, we approve the execution of the current query. If there are more than 10 similar queries, we deny the query and create a severe alert that is shown to the data officer via the PolicyLiner DemoUI, as depicted in Figure 2. If there are more than three similar queries but fewer than 10, we approve the execution of the query, but mark the query as *modified*, as we change its anonymization technique to static masking, which means that any data masking done to the query results is the same, thus making the query replay attack effectively impossible. Lastly, if there are fewer than three similar queries, we mark the query as *suspect* and create an alert warning for the data officer to take a closer look at the user and their queries.

Algorithm 2 Online Query Auditing Overview

Require: Disclosure query request Q_{DTO} with query string q, user id u, user role r, comparator type c

$queryUser \leftarrow$ find user by id u
if $queryUser$ does not exist **then**
    create new user U with id u and role r from Q_{DTO}
    $query.status \leftarrow APPROVED$
end if
$userQueries \leftarrow$ all queries by user u
if $userQueries$ is empty **then**
    $query.status \leftarrow APPROVED$
else
    if $|userQueries| > 1$ **then**
        $similarQueries \leftarrow \emptyset$
        for all q_{prev} in $userQueries$ **do**
            if $c = CUSTOM$ **then**
                $diff \leftarrow$ compute custom query comparison
                if $diff < 0.3$ **then** add q_{prev} to $similarQueries$
                end if
            else if $c = STRING$ **then**
                if q_{prev} equals q **then** add q_{prev} to $similarQueries$
                end if
            else if $c = LEVENSHTEIN$ **then**
                $sim \leftarrow 1-$ compute levenshtein distance percentage
                if $sim \geq 0.7$ **then** add q_{prev} to $similarQueries$
                end if
            end if
        end for
        if $similarQueries$ is empty **then**
            approve q
        else if $|similarQueries| \geq 10$ **then**
            deny q, and create severe alert to warn of query replay attack
        else if $3 \leq |similarQueries| < 10$ **then**
            implement static masking
            mark query as MODIFIED and create warning alert
        else
            mark query as SUSPECT and create warning alert
        end if
    else
        mark query as APPROVED
    end if
end if

Custom Query Comparison Implementation

In this section, we would like to talk more in depth about our custom query comparison algorithm and how we implemented it. We see the general implementation details of the custom query comparison in Algorithm 3. The return value of this algorithm is the difference ratio between the two queries in comparison and is always a value between 0 and 1. 1 means that the queries are totally different from one another, and 0 means that they are completely the same.

Algorithm 3 Custom Query Comparison Algorithm

Require: Previous query q_{prev}, current query q_{curr}, current table set T_{curr}
Ensure: Difference ratio in $[0, 1]$

if $q_{prev} = q_{curr}$ **then**
    return 0.0
end if
extract tables from q_{prev} and q_{curr}
$T_{prev} \leftarrow$ table names from q_{prev}
$differentTables \leftarrow \{t \in T_{curr} \mid t \notin T_{prev}\}$
$r_{table} \leftarrow \frac{|differentTables|}{|T_{curr}|}$
if $r_{table} = 1$ **then**
    return 1.0
else
    $r_{col} \leftarrow 0$; $r_{where} \leftarrow 0$
    extract column segments from q_{prev} and q_{curr}
    $C_{prev} \leftarrow$ columns of q_{prev}
    $C_{curr} \leftarrow$ columns of q_{curr}
    $differentColumns \leftarrow \{c \in C_{curr} \mid c \notin C_{prev}\}$
    if $differentColumns$ not empty **then**
        $r_{col} \leftarrow \frac{|differentColumns|}{|C_{curr}|}$
    end if
    if q_{prev} contains WHERE **and** q_{curr} contains WHERE **then**
        extract WHERE clauses W_{prev} and W_{curr}
        if $W_{prev} \neq W_{curr}$ **then**
            $differentWhere \leftarrow \{w \in W_{curr} \mid w \notin W_{prev}\}$
            if $differentWhere$ not empty **then**
                $r_{where} \leftarrow \frac{|differentWhere|}{|S_{curr}|}$
            end if
        end if
    end if
    return $\frac{r_{col}+r_{table}+r_{where}}{3}$
end if

There are three core values we keep track of in this algorithm:

- ratio of different tables (r_{table})
- ratio of different columns (r_{col})
- ratio of different conditions (r_{where})

The comparison first starts by naively comparing the query strings with one another. If the strings are completely the same, we return the value 0, as that means that the queries are actually 100% the same. In the likely event that the query strings are not exactly the same, we continue the comparison by extracting the tables that are being called in each of the queries. We check how many of the tables differ from one another between the two queries, and then we measure the ratio of different tables divided by the number of tables in the current query (r_{table}). If the ratio is 1, that means that the two queries do not have any of the same tables, leading us to believe that their results are also completely different. In this case, we return the value one and do not continue any further with the comparison. Otherwise, we continue the algorithm by extracting the table columns selected in each of the queries. We find the different columns between the two queries and measure the ratio of different columns (r_{col}) in the same way as the tables before. Lastly, we extract the individual conditions in the where clause of the queries and continue the same procedure with finding the ratio of different conditions (r_{where}) in the queries. The last return value is an average of these three ratios: r_{table}, r_{col}, r_{where}. We decided to give the ratios the same weight as all three can change the comparison between two queries fully, so giving the same weight gave the best results, as we will see in Chapter 5.

3.3 Offline Analysis

We will now present the implementation for offline analysis in PolicyLiner. PolicyLiner offline analysis is executed in two different independent components, as we saw in Figure 2. As such, we implement *Offline Query Auditing* as well as *Offline Policy Auditing*. Both workflows are run independently from one another at different intervals that can be chosen by the data officer.

3.3.1 Offline Disclosure Query Auditing

We start by describing the Offline Query Auditing process. As shown in Figure 2, the component responsible for executing this workflow is the **Query Analyzer**. As this process is run periodically, the starting point of it is also **Query Analyzer**, contrasting the online query analysis process, which started when the user wished

to execute a query. In this process, we also do not deal with just one user's queries; rather, all of the users who have executed queries are analyzed.

Nonetheless, the aim of this auditing process is to detect and counter the *Query Replay Attack*, similarly to the online query analysis. The reason behind this decision is simple. The most accurate way of detecting if two queries are similar is to compare the results that they produce. However, this action can not be efficiently completed during the user query execution, as it is a very costly operation and it would lead to unreasonable wait times for the query response, leading to it being effectively unusable in practice. Thus, we split the query comparison into two parts. One is the fast, efficient, and admittedly more error-prone comparison in the online query analysis. The goal of this process is to immediately detect if the user is trying to run malicious queries and to stop them. The other process is slower and requires more processing power, but leads to a supposedly more accurate comparison between queries. The objective of this offline analysis is to detect any historical queries that are malicious and inform the data officer of the suspect users and queries. In addition, the offline query analysis aims to inform the data officer of any false positive detections during the online query analysis.

We describe how we implement this analysis in detail in Algorithm 4. First, we start by getting a list of all users from the database. Then we loop through these users one by one. For each of the users, we get all the previously uninspected queries. *Uninspected* in our context signifies queries that have not yet been analyzed in our offline query auditing process. Afterwards, we loop through all these queries and compare them with each other.

At the start of this comparison, we get the status given by the online query analysis of the current query and check if it was deemed a suspicious query. We then obtain the tables of the two queries in comparison and find the common tables between them. If there are none, the comparison is skipped. Afterwards, we extract the columns from the two queries for comparison. Again, we detect if there are common table columns between the two, and if there are none, once more the comparison is skipped between these queries. These measures are taken to make the comparison as efficient as possible and not to go through unnecessary steps in finding the similarity between two queries that have none. Thereupon, we continue by rewriting the two queries to only contain the common columns and tables between them. Then we compare these two rewritten queries by running the *compareQueryResults* method in line 22 in Algorithm 4. This method executes both queries and compares their results. It then returns the sum of different rows between the two query results, as well as the sum of total rows of each of the query results. We then examine if the sum of different rows is 0. That means that the queries are equal. In this case, we mark both queries as suspect

Algorithm 4 Offline Query Auditing Overview

```
1: users ← list of all users
2: for all user ∈ users do
3:     Q ← list of all uninspected queries of user
4:     for i ← 0 to |Q| − 1 do
5:         q_curr ← Q[i]
6:         wasSuspect ← (q_curr status is SUSPECT, MODIFIED, or DENIED)
7:         isApproved ← true
8:         for j ← i + 1 to |Q| − 1 do
9:             q_prev ← Q[j]
10:            obtain table sets T_prev and T_curr from both queries
11:            T_common ← T_prev ∩ T_curr
12:            if T_prev ≠ T_curr and T_common is empty then
13:                continue
14:            end if
15:            extract column sets C_curr and C_prev
16:            C_same ← C_curr ∩ C_prev
17:            if C_same is empty then
18:                continue
19:            end if
20:            rewrite both queries using only columns in C_same
21:            let rewritten queries be q_1 and q_2
22:            (diffCount, totalCount) ← compareQueryResults(q_1, q_2)
23:            diffSum ← sum of differing rows
24:            totalSum ← sum of total rows
25:            if diffSum = 0 then
26:                mark both queries as SUSPECT
27:                isApproved ← false
28:                create severe alert to inform that queries are equal
29:            else
30:                ratio ← diffSum/totalSum
31:                if ratio ≤ 0.5 then
32:                    mark both queries as SUSPECT
33:                    isApproved ← false
34:                    create warning alert that queries are similar
35:                end if
36:            end if
37:        end for
38:        if isApproved and wasSuspect then
39:            create info alert of false positive from online analysis
40:            set q_curr status to APPROVED
41:        end if
42:        mark q_curr as INSPECTED
43:    end for
44: end for
```

and create a severe alert to inform the data officer. When that is not the case, we measure the ratio of different rows from the total rows of the query results. If this ratio is less than 0.5, we mark both queries as suspect and create a warning alert for the data officer as the queries are deemed to be similar to one another. In the end, we inspect if the current query was previously marked as suspect and was now approved by the offline analysis, and subsequently create an alert to inform the data officer of this false positive detection by the online analysis.

3.3.2 Offline Disclosure Policy Auditing

The following section examines the responsibilities and operational logic of the **Policy Analyzer** component. This component only operates during offline auditing and does not fulfill any assignments during online auditing. In addition, it is responsible for detecting and preventing different privacy attacks than the previous component. It aims to achieve this goal by implementing different privacy metrics that each detect a different attack on the primary database. These metrics are namely: *t-Closeness*, *δ-Presence*, *Sample Uniqueness Ratio* and *Population Uniqueness Estimation*. One other important feature of this component is its extensibility. If others want to extend the capabilities of the Policy Analyzer, they can easily add a new privacy metric to examine their disclosure policies, as the structure for implementing new metrics is already in place.

In order to use the capabilities of Policy Analyzer, we first need to configure some attributes. We start by defining the quasi-identifiers for each of the active disclosure policies, and then we continue by defining some thresholds for the privacy metric results we get from each of the privacy metrics. Additionally, we define the sensitive attributes of the disclosure policies that have them. The privacy metric thresholds define when the data officer must take action to improve data privacy or when the data officer has room to improve data utility in the primary database. Just as the extensibility mentioned above, these attributes are easily configurable by the user as well.

t-Closeness

Attribute Disclosure Attack aims to infer sensitive attributes from an individual. These attributes shouldn't be directly linkable to a person, as they reveal sensitive and private information about the person. This is where the *t-Closeness* privacy metric performs best. Its main goal is to prevent Attribute Disclosure Attacks[41]. Thus, we implemented the *t*-Closeness metric in our Policy Analyzer component.

We start by selecting the previously defined quasi-identifier attributes as well as the sensitive attributes of a policy view. We then loop through the sensitive attributes of a disclosure policy and measure the *t*-Closeness metric for each of them.

First, we measure the global distribution of the particular sensitive attribute and the proportion it takes in the whole disclosure policy view. Secondly, we calculate the distribution of the sensitive attribute in each of the equivalence classes. After gathering this information, we calculate the distance between each of the equivalence classes' distribution and the global distribution of the sensitive attribute. Here, we make a change to the original paper from Li et al. and use a different algorithm from the *Earth Mover's Distance (EMD)* Algorithm used by them [22]. We use an algorithm proposed by Dosselmann et al. called *Efficient EMD*, depicted in Algorithm 5, which, in contrast to the original EMD implementation, needs only one pass through the data, giving it a linear time complexity of $O(n)$, which is an improvement over the naive EMD implementation time complexity of $O(n^2)$ [10].

Algorithm 5 Efficient EMD [10]

1: $EMD \leftarrow 0$ ▷ Initialize EMD
2: $S \leftarrow 0$ ▷ Initialize current sum S
3: **for** $i = 1$ to m **do** ▷ For each $p_i \in P$ and $q_i \in Q$
4: $\quad S \leftarrow S + (p_i - q_i)$ ▷ Update cumulative difference
5: $\quad EMD \leftarrow EMD + |S|$ ▷ Accumulate absolute deviation
6: **end for**
7: $EMD \leftarrow \frac{EMD}{m-1}$ ▷ Valid for $m \neq 1$

The threshold proposed by literature is 0.2 [22, 39]. So that is what we have used here as the threshold for the disclosure policies to achieve in order for their data masking process to be successful. A value that is bigger than 0.2 triggers a warning alert to the data officer, recommending a review of the disclosure policy. In addition, we have set double this threshold, 0.4, as the smallest value where the data officer receives severe alerts that the policies are susceptible to attribute disclosure attacks and to immediately change the disclosure policies definitions. A value lower than 0.05, or one-fourth of the original threshold, is set as the biggest value, where a data utility improvement is described as possible to the data officer.

δ-Presence

Membership Disclosure Attacks aim to decipher if a certain individual is a part of a particular dataset. This attack can find important information about an individual depending on the sensitivity and type of the dataset. As such, it must be prevented. The δ-Presence privacy metric, introduced by Nergiz et al. [30], measures the vulnerability of anonymized datasets to membership disclosure attacks. δ-Presence is computed from the ratios between quantities of each combination of quasi-identifier attributes in a sample dataset and in the larger population dataset. The

δ value in the δ-Presence metric is the largest ratio between the two quantities across the dataset [9]. Nergiz et al. [30] in the original paper measure the largest ratio, as well as the smallest one. Thus the δ parameter actually contains two values: δ_{min} and δ_{max}. This was done to protect against a symmetric attack, meaning hiding that an individual is *not* part of the dataset[9].

The challenge with implementing the δ-Presence metric in practice is that it needs knowledge of the general population in order to be computed. Therefore, we need the sample dataset as well as a reliable population dataset to measure the δ value exactly. To address this challenge, Nergiz et al. proposed a new implementation in a follow-up paper [31]. This new implementation does not assume knowledge of the full population. It assumes we know distributions for attributes of the general population (e.g., the number of males in Berlin who are 25 years old), not individual data points. It also introduces c-confident δ-presence, which guarantees that the δ value holds with confidence of level c, given only the attribute distributions, not the full population data. Unfortunately, this implementation comes with three big problems when trying to implement it in practice.

- We do not have and cannot assume that the data officer using PolicyLiner has access to accurate distributions of attributes in the population.
- Computing the algorithm is very expensive for large populations and datasets. The authors propose a lot of approximations to make it feasible, but then, using them makes the results even more uncertain [9].
- Deciding and specifying a good confidence level c is also subjective and cannot be reliably defined, which lowers the dependability on the δ-presence metric.

On that account, we decided to implement our own δ-presence estimation metric, which calculates membership disclosure attack risk using *Good-Turing frequency estimation* and *Bayesian inference*. We estimate population frequencies of equivalence classes found in our primary database using the Simple Good-Turing estimator [14]. We decided to use this estimator as it is a general-purpose, well-researched, and widely used method that gives good results in most cases, even if it is not the most accurate algorithm in certain more specific situations [34, 35]. In the Good-Turing estimation technique, we adjust the likelihood of rare events appearing in the general population. When equivalence classes appear rarely in the sample dataset, we adjust their count in the following way:

For singletons:

$$adjusted\_count = 2 * N(2)/N(1) \tag{3.1}$$

For doubletons:

$$adjusted\_count = 3 * N(3)/N(2) \tag{3.2}$$

For larger frequencies:

$$adjusted\_count = (r+1) * N(r+1)/N(r) \tag{3.3}$$

where $N(r) = number\ of\ equivalence\ classes\ that\ appear\ exactly\ r\ times$

After estimating the frequency of equivalence classes in the general population, we measure the δ value or membership disclosure risk using the Bayes theorem and the definition of δ-Presence by Nergiz et al. mentioned in Chapter 2 [30]. In the Bayesian framework, we have two important concepts:

1. **Prior Probability**: In our situation, it means the adversary's belief that an individual is in our primary database, before seeing the disclosure policy view of the data. This is defined as:

   $$P(individual \in T) = |T|/|P| = sampling\_rate \tag{3.4}$$

   for

   - $T = primary\ database\ or\ dataset$
   - $T^* = anonymized\ version\ of\ T$
   - $P = public\ population\ dataset$

2. **Posterior Probability**: The adversary's belief that an individual is in our primary database after observing the anonymized version of the database/-dataset. This is defined as:

   $$P(individual \in T \mid T^*) = P(T^* \mid individual \in T) * P(individual \in T)\ /\ P(T^*) \tag{3.5}$$

   Using the law of total probability, we get:

   $$\begin{aligned} P(T^*) = &\ P(T^* \mid individual \in T)\ *\ P(individual \in T)\ + \\ &P(T^* \mid individual \notin T)\ *\ P(individual \notin T) \end{aligned} \tag{3.6}$$

   Substituting the following definitions, with labels that are easier to understand,

   - $estimated\_population\_frequency\ =\ P(T^* \mid individual \notin T)$
   - $observed\_sample\_frequency\ =\ P(T^* \mid individual \in T)$

   we get the following definition of Posterior Probability:

   $$\begin{aligned} &posterior\_probability = \\ &(sampling\_rate * observed\_sample\_frequency)\ / \\ &(sampling\_rate * observed\_sample\_frequency\ + \\ &(1 - sampling\_rate) * estimated\_population\_frequency) \end{aligned} \tag{3.7}$$

After measuring these two concepts, finding the δ value is simple. The equation is defined as follows:

$$\delta = |posterior\_probability \ - \ prior\_probability| \tag{3.8}$$

This metric measures how much the anonymized disclosure policy view changed an attacker's belief about the membership of an individual in the dataset.

The two minimum and maximum thresholds recommended by literature are $\delta_{min} = 0.05$ and $\delta_{max} = 0.15$ [30, 39]. That is why we have used them as well. When the δ value is not between these two thresholds, the data officer receives a warning alert to review and improve the data privacy in the particular disclosure policy in order to mitigate a membership disclosure attack. Additionally, we add a threshold of 0.3, as the value where a severe alert with an immediate action recommendation to the data officer is issued.

Sample Uniqueness Ratio

There are two types of uniqueness metrics as mentioned in Chapter 2: *Population Uniqueness Estimation* and *Sample Uniqueness Ratio*. Both metrics are aimed at preventing a *Re-identification Attack*, whose goal is to link an individual with a specific data row in the database. Sample Uniqueness Ratio is the easiest to implement and requires only knowledge of the sample dataset, while Population Uniqueness estimation requires in addition to the sample dataset, a population dataset.

Sample Uniqueness Ratio is implemented in a straightforward way. We first select all quasi-identifier rows from the particular policy, creating equivalence classes. We then obtain the frequency of each of these equivalence classes. Afterwards, we select and count how many of these equivalence classes are unique (have a frequency of 1) in the view used by the particular policy. Lastly, we measure the ratio between these unique equivalence classes and the whole size of the dataset, and we call that the sample uniqueness ratio.

The best theoretical threshold for such a metric is naturally 0. So that is the goal we aim for disclosure policies to achieve in order for their data masking process to be successful. In addition, a sample uniqueness ratio of 0.01 triggers a severe alert that calls for immediate action by the data officer. On the opposite side, we cannot set a sample uniqueness ratio lower threshold that would allow for more data utility. For that reason, there is no lower threshold for this metric.

Population Uniqueness Estimation

In view of the fact that the *Sample Uniqueness Ratio* might not be enough to confidently address the risk of a *Re-identification Attack*, we implement the *Population Uniqueness Estimation* as well. We follow the research of Dankar et. al.

and implement the decision algorithm they created in order for the population uniqueness estimation to be as accurate as possible [8]. As mentioned in Chapter 2, Dankar et al. [8] use three different uniqueness estimator algorithms in different situations: Pitman [36], SNB[7], and Zayatz[53]. We implement these three estimators and the decision rule depicted in Algorithm 1. The combination of the two uniqueness metrics gives us a better overview of the *Re-identification Attack* risk in each of the disclosure policy views.

The same theoretical threshold for the Population Uniqueness Estimation, just as the Sample Uniqueness Ratio, is, of course, 0. This is, however, not feasible in practice, as this metric only estimates the population uniqueness ratio; it cannot calculate it exactly due to the fact that we cannot assume that the data officer in a real setting has access to a reliable population dataset. Therefore, we aim for a threshold of 0.01 population uniqueness ratio as the goal that the data masking process of the disclosure policies should achieve. Additionally, a population uniqueness ratio of 0.1 triggers a severe alert that calls for immediate action by the data officer, and a low threshold does not exist.

Disclaimer. The Population Uniqueness Estimation was not thoroughly tested and might not be 100% correct in its implementation. The research of Dankar et. al. was followed, and the individual algorithms were implemented to the best of our ability, but they could contain discrepancies from the original implementation of Dankar et. al.[8]

4 Implementation

In this chapter, we would like to explain in detail how we implemented the concepts and components described in chapter 3. We will go in-depth about how the PolicyLiner software architecture is built, the technologies that were used, and how every component, service, and resource is organized. Furthermore, we will describe the PolicyLiner DemoUI implementation, its capabilities, and usefulness in testing and showcasing the theoretical concepts described in chapter 3. The whole implementation is stored in a public GitHub repository:
`https://github.com/irhox/policyliner`

4.1 PolicyLiner

Although PolicyLiner was created first and foremost as a database auditing tool of disclosure policies and queries created by Mascara, we implemented it as a standalone application that can theoretically be used by any other system. PolicyLiner has a REST API that can be used by any other system as long as they adhere to the requirements of the individual endpoints. Thus, PolicyLiner can be an open-source application that can be used by all developers and systems that need a database privacy auditing tool. We created PolicyLiner in the Java programming language with the Quarkus framework [42]. We used Quarkus as it is a modern open-source framework that can be easily deployed to the cloud and offers great performance and developer experience [33].

4.1.1 Configuration

In this section, we will describe all the processes we implemented that need to be completed in order for the auditing processes to work correctly. First, the masking functions must be defined in the primary database. We created some additional masking functions in our GitHub repository, which were used during our experiments in chapter 5. These functions need to be stored in the database for the data officer to be able to create the disclosure-compliant policies. Secondly, the default metrics that act as thresholds for our alerts during the offline auditing process need to be defined. We already defined them for our experiments, but they can be easily configured by other users who might want stronger or looser privacy

guarantees. These metrics are described in the *default_metrics.json* file in the resources folder in our project. Now we describe the first part of our rest API, the Policy Administrator component mentioned in figure 2. We started developing PolicyLiner by first creating the Policy Administrator component to be able to define disclosure-compliant policies directly from PolicyLiner. For this purpose, the *PolicyResource* was created, which contains all rest API endpoints that are connected to disclosure policies. Two of those endpoints manage the parsing of data mask statements into SQL statements.

- ***/api/policy/create/object***: This endpoint's mission is to create disclosure policies that are defined in the request object of this endpoint. It was created with wide compatibility in mind. Other systems besides Mascara can use this endpoint to create their disclosure policies without having knowledge of SQL or the data mask language from Mascara. The request object is shown in code fragment 6. It contains all the needed information about the definition of a disclosure policy. The *tables* array contains information about the tables that will be used in this policy and their primary and foreign keys. The *columns* array defines information about what columns from the aforementioned tables are used, as well as the masking functions that need to be used. The *userRole* contains the allowed user role that can access this disclosure policy. The *useDefaultMetrics* boolean is used to store the default metrics defined in the *default_metrics.json* file in the database in connection with the newly created disclosure policy. The *useStaticMasking* defines the way we store the disclosure policy in the database, in a normal view (false) or a materialized view (true). Lastly, we have *evaluatePolicyUponCreation*, a boolean attribute, which, when true, runs the offline policy auditing process only for this newly created policy directly after it is created.

Code Fragment 6: Policy Creation DTO

```
{"createPolicyDTO": {
        "policyName": "string",
        "tables": "TableInfo[]",
        "columns": "ColumnInfo[]",
        "userRole": "string",
        "useDefaultMetrics": "boolean",
        "useStaticMasking": "boolean",
        "evaluatePolicyUponCreation": "boolean",
        "quasiIdentifier": "JsonQuasiIdentifier"
    }
}
```

To be able to evaluate this policy, we need the quasi-identifiers and sensitive attributes from it, and that is the role of the *quasiIdentifier* object seen last in code fragment 6.

This request object is sent to the *PolicyService* class, where the *createPolicy* method is executed. This method first parses the SQL statement from the *createPolicyDTO*. It then stores the disclosure policy as a view or materialized view in the primary database, depending on the option chosen for the attribute *useStaticMasking*. Afterwards, the disclosure policy object, mentioned in code fragment 4, is created and stored in the PolicyLiner database. If the *useDefaultMetrics* attribute is true, all the metrics from the *default_metrics.json* file are stored in the PolicyLiner database and related to the new disclosure policy object.

- ***/api/policy/create/query-string***: This endpoint aims to create disclosure policies using data mask statements. The request object of this endpoint is shown in code fragment 7. The difference between this request object and the one shown in code fragment 6 is the *policy* attribute. This attribute contains the data mask statement that defines the disclosure policy and, as such, requires no *table*, *columns*, or *policyName* attributes as the previous request object. The other attributes are the same as described in the endpoint above.

Code Fragment 7: Data Mask Policy Creation DTO

```
{"createPolicyFromStringDTO": {
        "policy": "string",
        "useDefaultMetrics": "boolean",
        "useStaticMasking": "boolean",
        "evaluatePolicyUponCreation": "boolean",
        "quasiIdentifier": "JsonQuasiIdentifier"
    }
}
```

In contrast with the previous endpoint, this one starts its implementation by parsing the data mask statement into the request object in code fragment 6. This work is done in the *parseDisclosurePolicyStatement* method in the *PolicyService* class. Afterwards, the workflow continues on the same path as the endpoint above. This was done to minimize code duplication and keep all pieces of code organized and up-to-date during development.

The *Policy Administrator* component contains another endpoint in addition to the other two.

- ***api/policy/deactivate/policyId***: This endpoint has the goal of deactivating out-of-date disclosure policies. It uses the *policyId* path parameter to find and select the needed disclosure policy, and then, with the use of the *deactivatePolicy* method in the *PolicyService* class, it deactivates it, records the time of this adjustment, and lastly stores the changes to the PolicyLiner database.

The last part of the configuration workflow is the definition of the quasi-identifiers for the already created disclosure policies in the previous step. This can easily be done in the *quasiIdentifiers* folder by creating a JSON file in the form of our already created *emr_x_patients_db.json* file. This file can act as a template for other databases and quasi-identifiers that the users of PolicyLiner might operate with. The path of the new quasi-identifiers file should be added to the *application.properties* file as the value of the *policyLiner.quasi-identifiers.file-path* variable in order for it to be used by PolicyLiner.

4.1.2 Policy Analyzer

Offline Policy Auditing

After completing the configuration workflow, we have the capability of running the offline policy auditing process. We configure the intervals between each auditing process by defining the *policy.evaluation.interval* variable in the *application.properties* file. The auditing process starts by first obtaining the active disclosure policies from the PolicyLiner database. Afterwards, we run a loop that evaluates each of the policies one by one. Inside of this loop each one of our privacy metric services evaluates the policy and creates the appropriate alerts if necessary. These services are defined in the same way and follow the structure of a common interface called *PrivacyMetricService*. The structure of this interface is defined in code fragment 8. The first method in this interface, *computeMetricForTable* implements the code for computing the score of the privacy metric when the disclosure policy is evaluated against it. The *computeMetricForTables* does the same thing but for multiple policies at the same time. The *initializeQuasiIdentifiers* method is already implemented and only needs to be called when we need to obtain the defined quasi-identifiers and sensitive attributes from the quasi-identifiers JSON file. The *evaluatePolicyAgainstMetric* method receives the privacy metric score from the *computeMetricForTable* method and decides if this score warrants alerts to the data officer or not.

This interface was created to make it easier for developers and users of PolicyLiner to extend the auditing capabilities of PolicyLiner. If another developer wants to use a different metric for their system in addition to the ones already

implemented, they can easily do this in PolicyLiner by implementing the *PrivacyMetricService* interface and using that new metric service in the *PolicyService* class with all the others. It is just a matter of adding a new line of code that calls the *evaluatePolicyAgainstMetric* method of the new privacy metric service inside the *evaluateDisclosurePolicies* method in the *PolicyService* class.

Code Fragment 8: Privacy Metric Interface

```
public interface PrivacyMetricService<T> {
    T computeMetricForTable(
        String tableName,
        JsonQuasiIdentifier localQuasiIdentifier);

    List<T> computeMetricForTables(
        List<String> tableNames,
        JsonQuasiIdentifiers localQuasiIdentifiers);

    void evaluatePolicyAgainstMetric(
        Policy policy,
        JsonQuasiIdentifiers localQuasiIdentifiers);

    default JsonQuasiIdentifiers
       initializeQuasiIdentifiers(ObjectMapper
       objectMapper) {
        try (InputStream is =
           getClass().getClassLoader()
            .getResourceAsStream(
            "policyLiner.quasi-identifiers.file-path"))
               {
            return objectMapper.readValue(is,
               JsonQuasiIdentifiers.class);
        } catch (Exception e) {
            throw new RuntimeException("Failed to
               load JSON quasi-identifiers file",
               e);
        }
    }
}
```

In addition to the scheduled method of offline policy auditing, we make a new endpoint for manually starting the auditing process. This endpoint, */api/policy/-analyze*, calls the same method in the *PolicyService* class as the scheduled process, but it is created for convenience and testing purposes.

The offline policy auditing process runs on its own virtual thread, so its performance is unimportant to the general availability of the PolicyLiner system, as it can fulfill other requests during the scheduled offline policy auditing process. Multi-threading is implemented using the virtual thread feature from Quarkus and Java 21[43]. It was the preferred method used, as it is lightweight and fairly easy to implement in practice. We deemed it a necessary feature for PolicyLiner to enable users to schedule the offline policy auditing processes at their desired intervals without dwelling on the implications of the general performance and availability of their system to other requests.

Miscellaneous

The last couple of endpoints that are part of the *Policy Analyzer* component and at the same time the *PolicyResource* class are:

- ***/api/policy/policyId***: This endpoint finds the disclosure policy using the *policyId* path parameter and returns all the information saved in the PolicyLiner database about the chosen policy. Additionally, the alerts created in connection with this policy are also part of the response.
- ***/api/policy/search***: This endpoint searches for all disclosure policies that fulfill a certain search filter and returns a paginated list of the same response object as the endpoint above.

4.1.3 Query Analyzer

Online Query Auditing

The online query auditing process is started by calling the first endpoint in the *QueryResource* class: ***/api/query/analyze***. This endpoint uses the object described in code fragment 5 as its request body. It sends this object to the *QueryService* class, where it is analyzed, and the response that is displayed in code fragment 9 is returned.

Code Fragment 9: Online Query Auditing Response DTO

```
{"queryResponseDTO": {
        "id": "string",
        "userId": "string",
        "query": "string",
        "status": "enum",
        "inspectionStatus": "enum",
        "message": "string",
```

```
8               "createdAt": "localdatetime",
9               "alerts": "Alert[]"
10          }
11      }
```

This response object contains the query id; the id of the user that executes the query; the query string that may have been modified in the analysis process; the status of the analysis, the inspection status, which shows us if the query has been analyzed by the offline query auditing process; a message that indicates the findings of the online auditing process; the time this query was executed and all the alerts created in connection to this query; in this order.

With this information, the other system that calls this endpoint can make an informed decision about allowing or denying access to the user who wants to execute their query.

Offline Query Auditing

The offline query auditing process is a scheduled task just as the offline policy auditing process. The variable that defines the interval of each execution is the *query.evaluation.interval* variable in the *application.properties* file. Similar to the offline policy auditing process, the offline query auditing process runs on a different virtual thread that does not interfere with the offline policy auditing process, as well as the other requests to PolicyLiner. Multithreading, in this case, is implemented in a similar way to the offline policy auditing process. The process is implemented on the *QueryService* class inside of the *offlineQueryAnalysis* method.

Miscellaneous

The last couple of endpoints for the Query Analyzer component and the *QueryResource* class are similar to the Policy Analyzer component.

- ***/api/query/queryId***: This endpoint finds the disclosure query using the *queryId* path parameter and returns all the information saved in the PolicyLiner database about the chosen disclosure query in the form of the object displayed in code fragment 9.
- ***/api/query/search***: This endpoint searches for all disclosure queries that fulfill a certain search filter and returns a paginated list of the same response object as the endpoint above.

4.1.4 Database

The database technology we used to store all the information of the PolicyLiner workflows, as well as the primary database, was PostgreSQL. We kept the primary database and PolicyLiner database detached from one another to have a clear separation of concerns and for the PolicyLiner system to be able to write only on the PolicyLiner database, while keeping the primary database read-only and unchanged.

While PolicyLiner was only tested with PostgreSQL, it should be compatible with most mainstream relational databases as it uses common SQL syntax between them.

4.2 PolicyLiner DemoUI

PolicyLiner DemoUI is our way of showcasing the functionality of PolicyLiner in an easily understandable and usable form. It is created using Typescript and the Angular framework version 20 [15].

4.2.1 Policies View

In figure 3, we see the home page of the demoUI, where all the policies, active and inactive, are displayed.

Figure 3: Home Page with Policies Overview

The information displayed on the individual cards for each of the policies contains the policy name, ID, status, and the time and date it was created. Additionally, each of the cards contains a button called "details" that, when clicked, displays detailed information about the policy on a new page. The home page is completed with a search bar to make it easier to find the needed disclosure policy, as well as pagination.

Policy Details

After clicking the details button of a policy, we find ourselves in the page displayed in figure 4. Here we can explore the disclosure policy creation query and the allowed user roles, in addition to the information shown in the overview page.

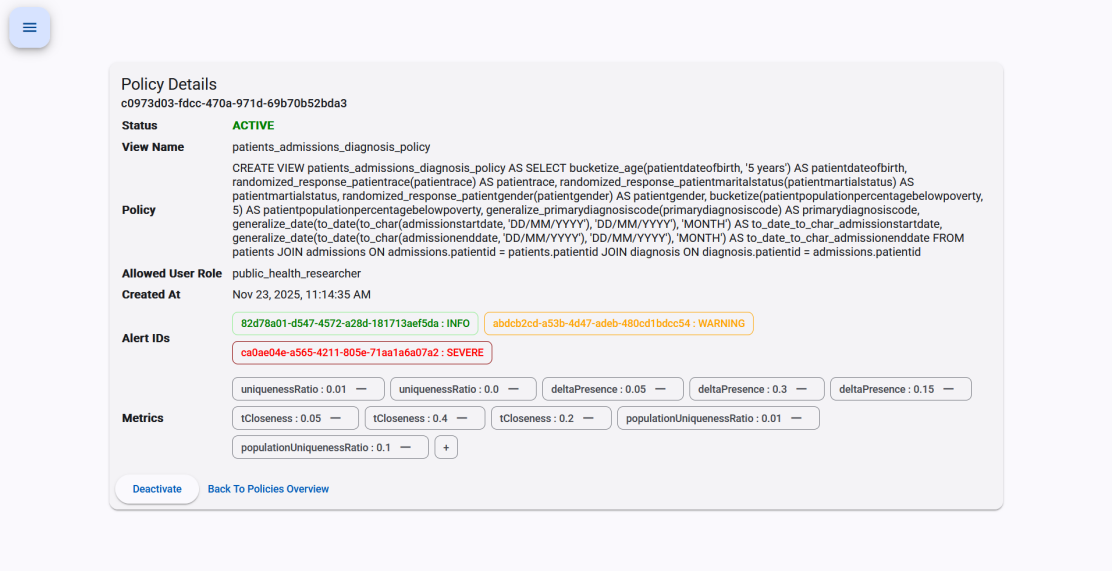

Figure 4: Policy Details View

Moreover, we can see the alert IDs connected to the particular policy as well as their severity, which, when clicked, show us the details of the individual alerts. In a similar fashion, we can see the metrics that are connected to the individual policy under the alerts. We can click those metrics to view their details, as shown in figure 5, or we can click the − button to delete them. To add new metrics, we can click the + button. Lastly, we can deactivate the policy by clicking the button at the bottom left of the page.

Privacy Metric Details

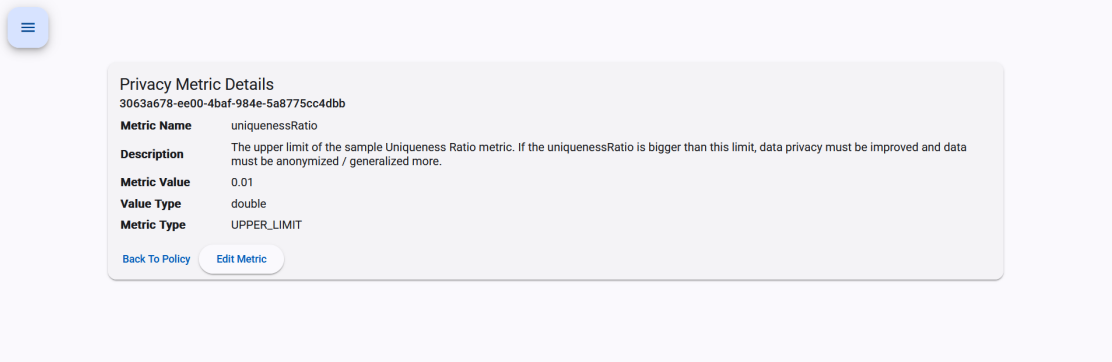

Figure 5: Privacy Metric Details View

After clicking the metric we want to view, we come to the page shown in figure 5. It displays the name, value, description, and type of the privacy metric. We can edit the details of the privacy metric when we click the edit button.

Policy Creation View

Going to the next menu point on the sidebar, we can find the policy creation form displayed in figure 6. It contains the text input field, where the data mask statement for defining the disclosure policy is written.

Additionally, there are three options that change the way the policy is submitted and what happens after. When using the default metrics option, we store the default metrics defined previously in the *default_metrics.json* file, in the database to use them when the offline policy auditing process is executed. The static masking option defines the way the data masking functions will work. If the option is not active, the policy user-defined functions can produce varying results every time a query is run on that policy. On the opposite side, if static masking is active, the user-defined functions return the same result always. The last option lets us decide if we want to evaluate the disclosure policy upon creating it or not. In the case that this option is active, more input fields are displayed on the form to define the quasi-identifiers and sensitive attributes of the disclosure policy.

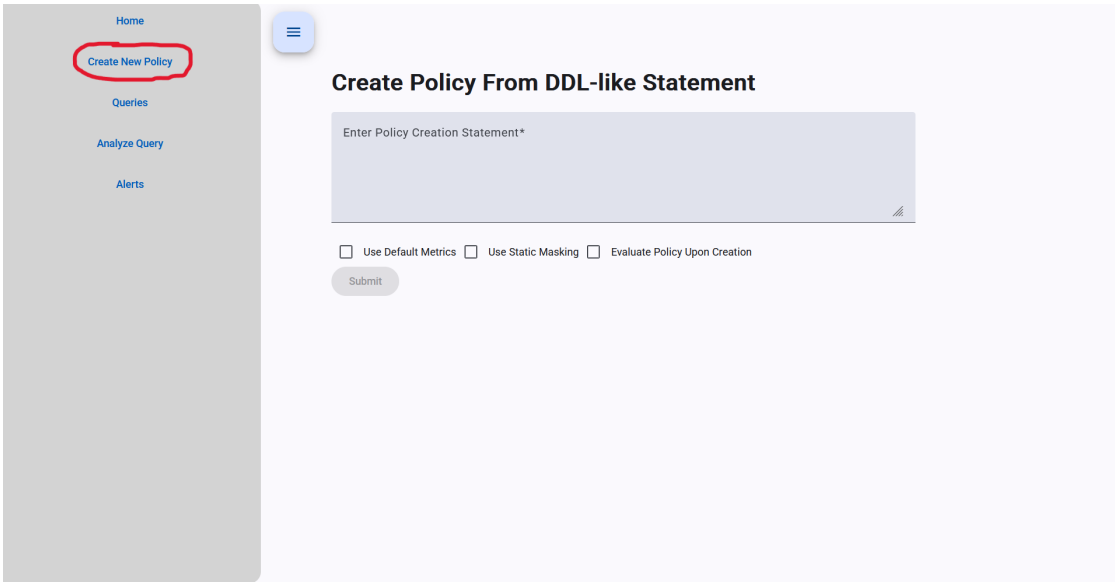

Figure 6: Policy Creation View

4.2.2 Queries View

In figure 7, we can view all the previously run queries in our system. The page is constructed in a similar way to the policies overview page in figure 3. The individual cards contain the query ID, status, the user ID that is executing them, and the time they are executed. The "Details" button has the same behavior in this page as the policies page; it shows the user the query details.

Query Details

When accessing the details page for an individual query, as displayed in figure 8, we can see the query ID, status, inspection status, the execution date, as well as the message sent to Mascara (or any other system that chooses to work in connection with PolicyLiner) after the online query auditing process for that query. Additionally, we display the various alerts for the query and make each of them clickable in order to view their details.

Query Analysis

In the next menu point in the sidebar, we view the query analysis form. On this page, we can simulate a query execution in real time and trigger the online query auditing process. Figure 9 presents the form with all its fields.

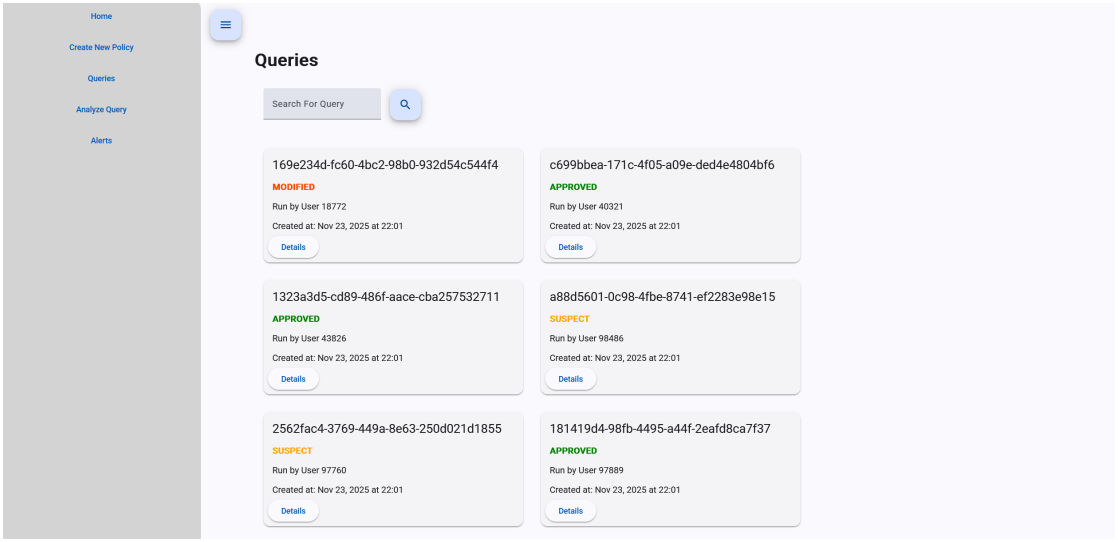

Figure 7: Queries Overview

Figure 8: Query Details View

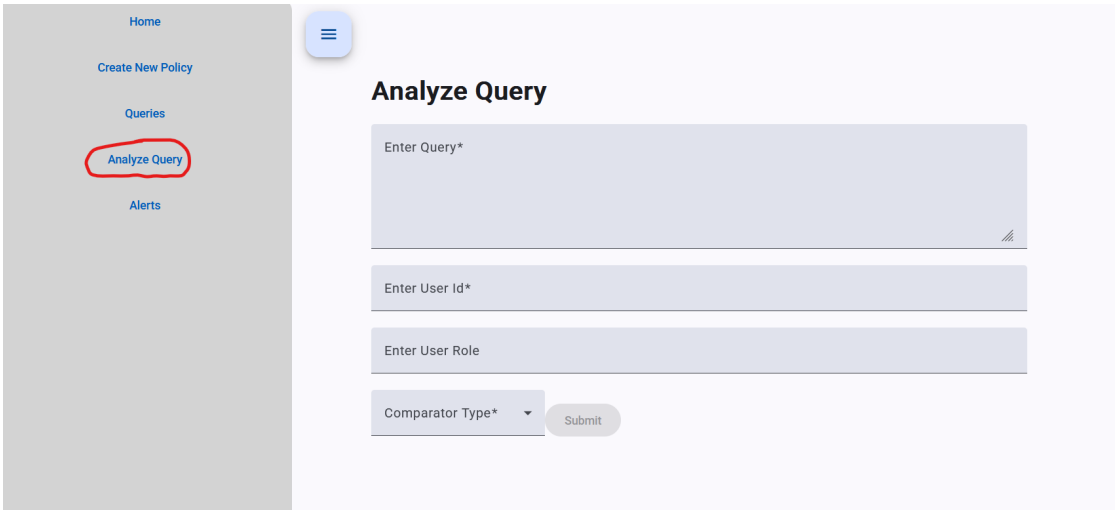

Figure 9: Query Analysis Form

In the first field, we write the actual query that we want to execute. In the second and third input fields, we define the user who is executing this query and the role he has in the system. Lastly comes the type of comparator we are using to define and evaluate if the user has executed similar queries before in our system. After filling all the input fields and submitting the form, we immediately get the response from PolicyLiner with the decided status and message of the analysis.

4.2.3 Alerts View

The last menu point of the sidebar shows us all the alerts created during the auditing processes. In figure 10, we see this page depicted and can discern that this page is structured in the same way as the other two overview pages of the queries and policies. An alert card on this page contains information about the kind of alert it is, the severity, and when it was created. It also displays the information if the particular alert was resolved and the issue was fixed, or if it still needs to be handled. At the bottom of each card, we can observe two buttons. The first one resolves the alert and tells us that the issue has been addressed, while the second one guides us to the alert details page.

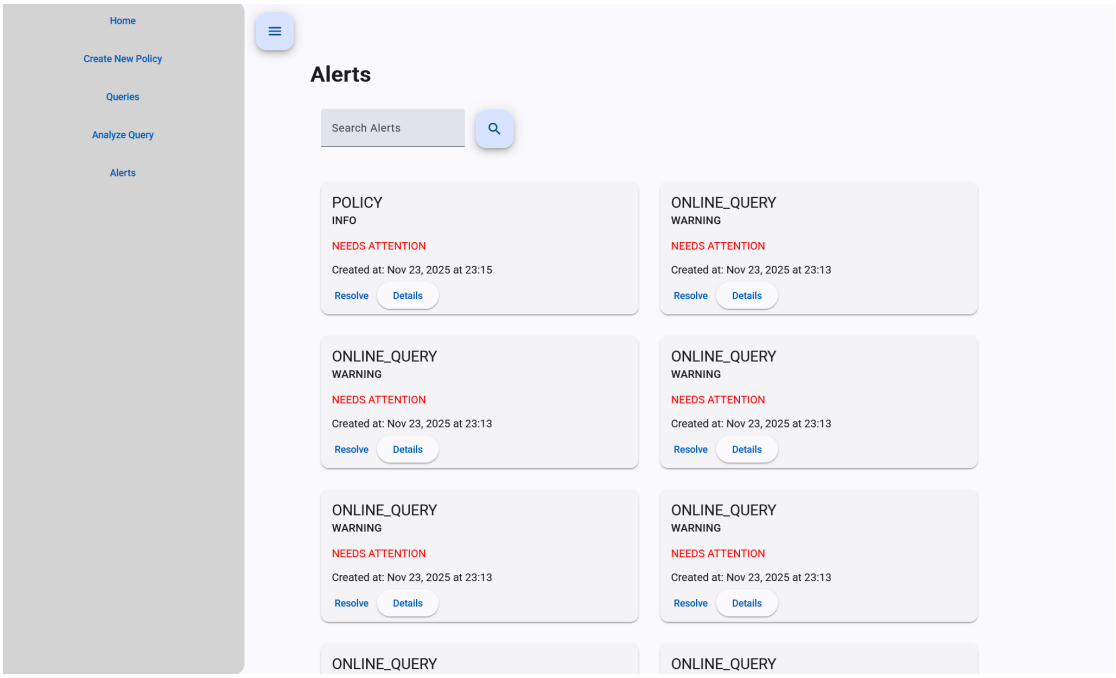

Figure 10: Alerts Overview

Alert Details

In figure 11, we showcase the alert details page. The information displayed on this page includes the alert ID, type, severity, and creation time. We can further explore the message of the alert as well as the status of its resolution. If this alert has been resolved, the time it was resolved is also displayed. Lastly, the policy or query IDs to which the alert is connected are depicted to make the problem it wants to give attention to as clear as possible. Just as in the previously mentioned details views, these IDs are also clickable and send the user to the respective details view.

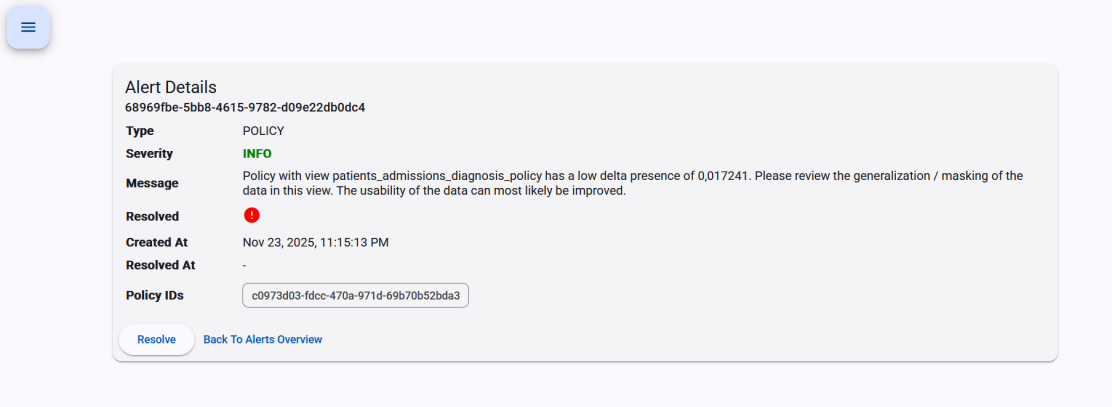

Figure 11: Alert Details View

5 Evaluation

In this chapter, we would like to show the results of the experiments that were executed on PolicyLiner. We evaluate the capabilities of PolicyLiner for recognizing dangerous scenarios when privacy attacks can be permitted by the disclosure policies and assess the performance of our system in real-time with online auditing and with periodically-run workflows with offline auditing. First, we create scenarios in which a privacy attack could happen in a practical setting and showcase how PolicyLiner can prevent such an attack from happening. Secondly, we evaluate the query comparison feature of our *Online Query Auditing* process. Lastly, we measure the overhead our auditing processes have on the main system and decide if the latency and storage requirements make PolicyLiner too inconvenient to use in practice.

We carried out these experiments with these questions in mind:

- Can PolicyLiner handle the data changes in databases with dynamic data?
- How accurate is our query comparison algorithm?
- Is our online query auditing process efficient enough to use in practice?
- Are our offline auditing processes efficient enough to use in practice?

5.1 Experimental Setup

Setup. We implemented PolicyLiner as described in Chapter 4. The whole PolicyLiner system was implemented in the Java programming language with the Quarkus framework [42]. For our performance experiments, we used Locust to simulate heavy traffic with many users to the system [17]. We performed these experiments on a portable computer with an AMD Ryzen 9 PRO 7940HS (4.0 GHz) processor with Radeon 780M Graphics and eight cores running Microsoft Windows 11. It also has 32 GB of memory and an SSD of 954 GB. For our offline auditing processes, we implemented a transaction timeout of 5 minutes, as we found that that was the optimal runtime limit for our hardware.

Metrics. For the query comparison experiments, we measure the Precision, Recall, and F1 Scores of each comparison algorithm. We also use correlation metrics and evaluate the accuracy of the different comparison algorithms.

For the performance experiments, we run the same evaluation multiple times and average the results to obtain an accurate, outlier-resistant outcome.

Datasets. Finding real public datasets with sensitive data is, by definition, something that should not be possible, and it is not easy to find. Therefore, to run our experiments, we used a generated dataset created by Kartoun et al. [20]. This dataset contains four tables: *patients*, *admissions*, *labs* and *diagnoses*. Their exact structure is shown in Table 1. We see that the information displayed in this synthetic dataset is similar to a sensitive medical dataset, which is applicable to the use case of PolicyLiner. Additionally, Kartoun et al. [20] provide three different sizes of this dataset, which they then call by the number of patient table entries: 100-patients, 10000-patients, and 100000-patients. This provides us with a variety of dataset sizes to compare the performance of PolicyLiner, as shown in table 2.

Patients	**Admissions**	**Diagnoses**	**Labs**
PatientID	PatientID	PatientID	PatientID
PatientGender	AdmissionID	AdmissionID	AdmissionID
PatientDateOfBirth	AdmissionStartDate	PrimaryDiagnosisCode	LabName
PatientRace	AdmissionEndDate	PrimaryDiagnosisDescription	LabValue
PatientMartialStatus			LabUnits
PatientLanguage			LabDatetime
PatientPopulationPercentageBelowPoverty			

Table 1: Dataset by Kartoun et al. [20]

	100-Patients	**10000-Patients**	**100000-Patients**
Patients	100	10000	100000
Admissions	372	36.143	361.760
Diagnoses	372	36.143	361.760
Labs	111.483	10.726.505	107.535.387

Table 2: Dataset table sizes

5.2 Effectiveness of Offline Policy Auditing

In this section, we will present different scenarios in which the effectiveness of PolicyLiner's offline policy auditing is exhibited. We will start by describing the

initial situation of the database as well as the disclosure policy definition. Then, we will simulate data changes that mimic the real world and display how the effectiveness of these disclosure policies can change very easily. The dataset used for these experiments was the **100000-Patients** dataset.

5.2.1 Scenario setup

Public health researchers want to access our database in order to study whether there are any correlations between diseases, the medical test results, and the patients' demographics. Upon receiving this request, the data officer creates a disclosure-compliant policy to serve the public health researchers with useful data, while keeping the personal data of the patients private. This policy is created with the query displayed in code fragment 10. As we can see, the policy contains a lot of private information attributes about the patients, their stay in the hospital, and their diagnosis. To protect the privacy of the patients, while giving useful aggregate data to the researchers, the data officer masks the attributes with the various user-defined masking functions and achieves the desired result.

Code Fragment 10: Public health researcher Disclosure Policy Definition

```
disclose patientdateofbirth, patientrace, patientmartialstatus,
    patientgender, patientpopulationpercentagebelowpoverty,
    admissionstartdate, admissionenddate, primarydiagnosiscode,
    from patients, admissions, diagnosis
    with mask on patientdateofbirth using bucketize_age('5 years')
    with mask on patientrace using
   randomized_response_patientrace()
    with mask on patientmartialstatus using
   randomized_response_patientmaritalstatus()
    with mask on patientgender using
   randomized_response_patientgender()
    with mask on patientpopulationpercentagebelowpoverty using
   bucketize(5)
    with mask on primarydiagnosiscode using
   generalize_primarydiagnosiscode()
    with mask on to_date(to_char(admissionstartdate,
   'DD/MM/YYYY'), 'DD/MM/YYYY') using generalize_date('MONTH')
    with mask on to_date(to_char(admissionenddate, 'DD/MM/YYYY'),
   'DD/MM/YYYY') using generalize_date('MONTH')
    where patients.patientid = admissions.patientid
    and admissions.patientid = diagnosis.patientid
    and $user.role = 'public_health_researcher'
```

The data officer then defines the quasi-identifier columns for this disclosure policy, which are: *patientdateofbirth*, *patientrace*, *patientmartialstatus*, and *patientgender*. Additionally, the *primarydiagnosiscode* column, which contains a code of the kind of disease the patient was diagnosed with, is defined as a sensitive attribute.

5.2.2 Initial State

To demonstrate that his disclosure policy is sufficiently private, the data officer examines the equivalence classes in the policy and the distribution of the sensitive attribute within them. There are 1080 equivalence classes in the disclosure policy with 1.540.586 entries total. The results, shown in table 3, indicate low risk of re-identification and membership disclosure attacks as the data is generalized in many different equivalence classes that have sufficiently big sizes [11, 12]. Table 3 in its third column also shows the distribution of sensitive attributes and how diverse it is in each of the equivalence classes. These results display adequate diversity of sensitive attributes that indicate low risk of attribute disclosure attacks [25, 52].

	Equivalence Class Size	**Sensitive Attribute Distribution**
Maximum	4229	364
Minimum	32	28
Average	1426,46	245,72
Median	1323	269

Table 3: Equivalence Classes Information

In addition, we run our offline policy auditing process and obtain the results shown in table 4, which strongly support the claims made above.

Metric	**Result**	**Thresholds**
δ-Presence	0,015	$\delta < 0,15$
t-Closeness	0,103	$t < 0,2$
Sample Uniqueness Ratio	0	0
Population Uniqueness Estimation	0,000701	$< 0,01$

Table 4: Initial Offline Policy Auditing Result

Having established that the created disclosure policy is compliant and protects the privacy of patient data appropriately, the data officer publishes this policy and allows the public health researchers to access it.

Goal. In the following subsections, we will explore real-world situations of data changes that can expose private information and make the disclosure policy non-compliant and obsolete, despite it being very well protected in its inception. Thus proving that an auditing tool like PolicyLiner is necessary in databases with constantly changing data.

5.2.3 Data Insertion

In the following weeks, after the creation of the disclosure-compliant policy for public health researchers, the hospital gets a new patient who is registered in the database with the following data shown in table 6. This patient contains outlier attributes as he is the only patient under thirty-five years old in the dataset. This means that a new equivalence class with just this new patient is created, leading to a full breakdown of the privacy guarantees mentioned in the scenario setup. Luckily, the data officer is using PolicyLiner and after the periodical offline policy auditing process, the results, as displayed on table 5, alert the data officer of the following risks:

- **Membership Disclosure Attack** warning alert is triggered by the δ-Presence score above the 0,15 value.
- **Re-identification Attack** warning alert is triggered by the sample uniqueness ratio that is bigger than 0. The population uniqueness estimation does not cross the threshold for alerting the data officer, as in a population of millions, it is very unlikely that a patient in their twenties is the only one with those particular quasi-identifiers, and that is reflected in the population uniqueness estimation score.
- **Attribute Disclosure Attack** severe alert is triggered by the t-Closeness score that is not only above the first threshold of 0,2 but also above the 0,4 severe threshold.

These attacks are all made possible by the fact that this new patient has a distinct age not covered by already existing age intervals, creating a unique entry that makes the patient identifiable, their membership in the dataset detectable, and their diagnosis discoverable.

Following the offline policy auditing process, the data officer can address the data changes by adjusting the disclosure policy to categorize the age of the patients into ten-year intervals and not five as was shown in the first definition of the policy in code fragment 10. While this reconstruction of the disclosure policy lowers the utility of the dataset to the public health researchers, it protects the data sufficiently again as shown by the offline policy auditing results in table 7.

Metric	**Result**	**Thresholds**
δ-Presence	0,1667	$\delta < 0,15$
t-Closeness	0,5446	$t < 0,2$
Sample Uniqueness Ratio	0,000001	0
Population Uniqueness Estimation	0,000702	$< 0,01$

Table 5: Offline Policy Auditing Result with New Patient Data

Patient Information	
PatientID	FB908FBC-73CD-5A6F-0821-T52183DF385F
Gender	Male
Date of Birth	1995-05-11
Race	White
Marital Status	Separated
Language	Icelandic
Poverty Rate (%)	18.31
Admission	
AdmissionID	10
Start Date	2025-06-25 09:21:00
End Date	2025-12-01 19:14:00
Diagnosis	
Primary Code	Z22.31
Description	Carrier of bacterial disease due to meningococci
Lab Results	
Lab Name	CBC: MCH
Lab Value	301
Lab Units	pg
Lab Datetime	2025-07-11 15:32:00

Table 6: New Patient Data

The summary of the difference in our privacy metrics as we move from the initial state, to the data insertion, and then the disclosure policy adjustment is displayed in figure 12. We can see the massive changes in the different situations in three of the four privacy metrics we have implemented. Furthermore, we can observe how one unique data point does not impact the population uniqueness estimation when the population estimation consists of millions of data points.

Metric	**Result**	**Thresholds**
δ-Presence	0,0139	$\delta < 0,15$
t-Closeness	0,1205	$t < 0,2$
Sample Uniqueness Ratio	0	0
Population Uniqueness Estimation	0,000374	$< 0,01$

Table 7: Offline Policy Auditing Results after policy adjustment

5.2.4 Data Deletion

We restart this scenario with the dataset and disclosure policy in their initial states. This time, we aim to showcase the dangers of data deletion to the privacy of sensitive datasets. Fortunately, this proved more challenging than the previous data insertion scenario. Thus, we simulate a scenario in which the hospital periodically cleans its database, depending on when patients were first admitted. We try different deletion periods: deleting all data before 1950, 1975, 1985, 1995, 2000, 2005, 2010, 2012, and 2015. The results are shown in Figure 13. The first fact we deduce from this experiment is that the distribution of quasi-identifiers and sensitive attributes in our dataset is very uniform and, as such, does not allow for the immediate selective deletion of data that could cause privacy breaches.

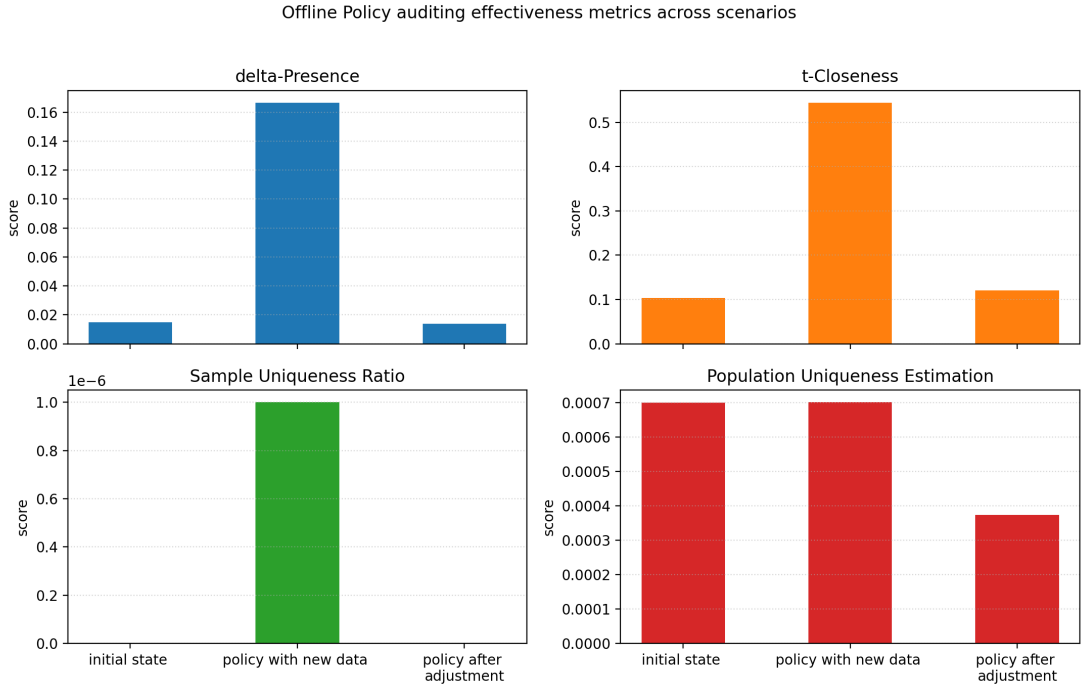

Figure 12: Disclosure Policy Privacy Measurements in different States

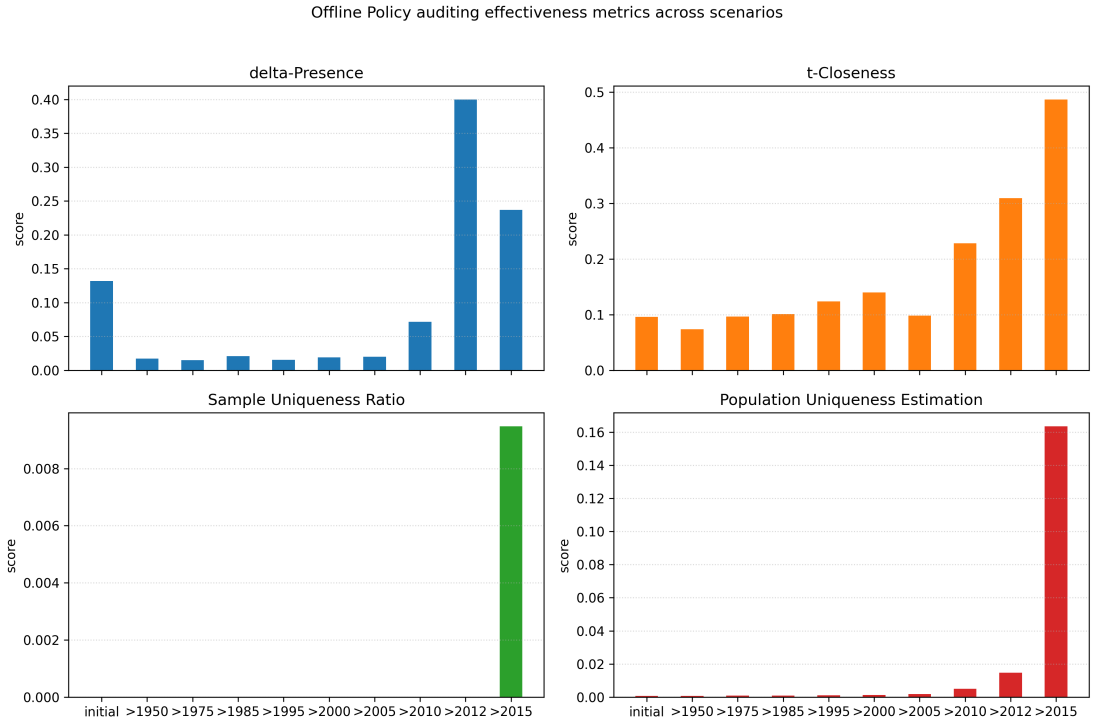

Figure 13: Disclosure Policy Privacy Measurements in different Deletion States

This is also apparent in the data size deleted after each cleanup action, as shown in table 8.

State	Disclosure Policy Size	Admissions Table Size
Initial	1.540.584	361.759
> 1950	1.513.097	355.785
> 1975	1.331.473	315.700
> 1985	1.180.721	281.513
> 1995	955.011	230.149
> 2000	800.523	195.815
> 2005	573.904	146.206
> 2010	216.695	60.017
> 2012	73.626	20.832
> 2015	422	422

Table 8: Dataset Sizes after Data Deletion in each Stage

The first admission in the dataset is in 1938, and the last is in 2017. Only after deleting every admission before 2010, that means about 83% of the admissions table is deleted, or 85% of the disclosure policy view entries, does the data officer get the first alert from one of the privacy metric evaluations, namely t-Closeness. After deleting all the admissions before 2012, we get alerts from all the privacy metric evaluations.

Takeaway. In practice, such a large, immediate data deletion situation would most likely not happen, but it displays the importance of uniform, well-distributed data, which in the real world is unfortunately also not common. Had the dataset not been well separated into similarly sized equivalence classes and had the distribution of sensitive attributes been poorly balanced, the results of our data deletion stages would most likely have looked very different and much worse.

5.3 Query Similarity Comparison

In this section, we would like to evaluate our custom query comparison algorithm used during our online query auditing process and find out if it is an accurate and error-free implementation, or if simple string comparison and the Levenshtein distance algorithm perform better in certain situations. We started this experiment by creating 48 sets of two or three queries. These sets are separated into groups:

- twelve sets contain two simple, similar queries inside

- twelve sets contain two or three more complex, similar queries inside
- twelve sets contain two simple, different, but similarly looking queries inside
- twelve sets contain two more complex, different, but similarly looking queries inside

We created these different sets to test whether our online query auditing process can not only recognize similar queries correctly, but also discern when similar-looking queries bring very different results. The exact query sets can be found in our open-source GitHub repository: `https://github.com/irhox/policyliner/tree/main/policyliner/src/main/resources/similarQueries`.

We ran these sets separately through our online query auditing process and got the following results displayed in tables 9, 10, 11, 12.

Tables 9 and 10 show us the accuracy of the different comparison methods in finding similar queries with similar or equal results when executed.

We can see from table 9 that our custom algorithm performs the best out of the three options, with 20/24 queries being recognized correctly. This is far better than the result of 12/24 from the Levenshtein distance algorithm. Going further into the table 10 of complex queries, the performance drops significantly for our custom algorithm, with only 17/30 complex queries being identified correctly, while for the Levenshtein distance algorithm, the performance drops only slightly to 14/30 or 46.7% of the queries being recognized correctly.

Table 9: Simple Similar Query Sets Comparison Result

Similar Set	**Custom Algorithm**	**Levenshtein Distance**	**String Comparison**
1	2/2 (100%)	0/2 (0%)	0/2 (0%)
2	2/2 (100%)	0/2 (0%)	0/2 (0%)
3	2/2 (100%)	2/2 (100%)	0/2 (0%)
4	2/2 (100%)	2/2 (100%)	0/2 (0%)
5	2/2 (100%)	2/2 (100%)	0/2 (0%)
6	2/2 (100%)	2/2 (100%)	0/2 (0%)
7	0/2 (0%)	2/2 (100%)	0/2 (0%)
8	2/2 (100%)	0/2 (0%)	0/2 (0%)
9	2/2 (100%)	0/2 (0%)	0/2 (0%)
10	2/2 (100%)	0/2 (0%)	0/2 (0%)
11	2/2 (100%)	2/2 (100%)	0/2 (0%)
12	0/2 (0%)	0/2 (0%)	0/2 (0%)
Total	**20/24 (83,4%)**	**12/24 (50%)**	**0/24 (0%)**

Table 10: Complex Similar Query Sets Comparison Result

Similar Set	Custom Algorithm	Levenshtein Distance	String Comparison
1	2/3 (66,7%)	2/3 (66,7%)	0/3 (0%)
2	3/3 (100%)	0/3 (0%)	0/3 (0%)
3	2/3 (66,7%)	2/3 (66,7%)	0/3 (0%)
4	3/3 (100%)	2/3 (66,7%)	0/3 (0%)
5	2/2 (100%)	2/2 (100%)	0/2 (0%)
6	0/2 (0%)	0/2 (0%)	0/2 (0%)
7	0/3 (0%)	0/3 (0%)	0/3 (0%)
8	0/2 (0%)	2/2 (100%)	0/2 (0%)
9	0/2 (0%)	2/2 (100%)	0/2 (0%)
10	2/2 (100%)	2/2 (100%)	0/2 (0%)
11	0/2 (0%)	0/2 (0%)	0/2 (0%)
12	3/3 (100%)	0/3 (0%)	0/3 (0%)
Total	**17/30 (56,7%)**	**14/30 (46,7%)**	**0/30 (0%)**

We can recognize a similar pattern of accuracy for the tables 11 and 12 that explore the other side of the coin, perceiving different queries that look similar correctly. The simple string comparison performs the best in both these cases, as all queries used in this experiment are different strings. This does not show good performance, but rather only the naive behavior of this implementation. The best out of the other two comparison algorithms is our custom algorithm with 20/24 or 83,4% accuracy in the simple different queries and a somewhat worse performance of 14/24 or 58,34% in the complex query comparisons. The Levenshtein Distance, on the other hand, performs really badly in both cases, with the simple different query comparisons result being 2/24 or 8,34% accuracy and the complex different query comparisons result being 6/24 or 25% accuracy.

Table 11: Simple Different Query Sets Comparison Result

Different Set	Custom Algorithm	Levenshtein Distance	String Comparison
1	2/2 (100%)	0/2 (0%)	2/2 (100%)
2	2/2 (100%)	0/2 (0%)	2/2 (100%)
3	2/2 (100%)	0/2 (0%)	2/2 (100%)
4	2/2 (100%)	0/2 (0%)	2/2 (100%)
5	2/2 (100%)	0/2 (0%)	2/2 (100%)
6	2/2 (100%)	0/2 (0%)	2/2 (100%)
7	2/2 (100%)	2/2 (100%)	2/2 (100%)
8	0/2 (0%)	0/2 (0%)	2/2 (100%)
9	2/2 (100%)	0/2 (0%)	2/2 (100%)
10	0/2 (0%)	0/2 (0%)	2/2 (100%)
11	2/2 (100%)	0/2 (0%)	2/2 (100%)
12	2/2 (100%)	0/2 (0%)	2/2 (100%)
Total	**20/24 (83,4%)**	**2/24 (8,34%)**	**24/24 (100%)**

Table 12: Complex Different Query Sets Comparison Result

Different Set	Custom Algorithm	Levenshtein Distance	String Comparison
1	2/2 (100%)	2/2 (100%)	2/2 (100%)
2	0/2 (0%)	0/2 (0%)	2/2 (100%)
3	0/2 (0%)	2/2 (100%)	2/2 (100%)
4	0/2 (0%)	2/2 (100%)	2/2 (100%)
5	2/2 (100%)	0/2 (0%)	2/2 (100%)
6	2/2 (100%)	0/2 (0%)	2/2 (100%)
7	2/2 (100%)	0/2 (0%)	2/2 (100%)
8	2/2 (100%)	0/2 (0%)	2/2 (100%)
9	0/2 (0%)	0/2 (0%)	2/2 (100%)
10	2/2 (100%)	0/2 (0%)	2/2 (100%)
11	0/2 (0%)	0/2 (0%)	2/2 (100%)
12	2/2 (100%)	0/2 (0%)	2/2 (100%)
Total	**14/24 (58,34%)**	**6/24 (25%)**	**24/24 (100%)**

To evaluate a general performance, we measure the Precision, Recall, and F1 Score of our findings displayed in table 13.

Table 13: Precision & Recall, F1 Score

	Custom Algorithm	Levenshtein Distance	String Comparison
Precision	37/51 (0,7254)	26/66 (0,394)	0/0 (not computable)
Recall	37/54 (0,6852)	26/54 (0,4814)	0/54 (0)
F1 Score	0,7047	0,4333	not computable

Lastly, we measured the Pearson correlation coefficient between the true similar queries quantity and the results of each of the comparison algorithms and displayed the results in figure 14 for the simple query results and in figure 15 for the complex query results.

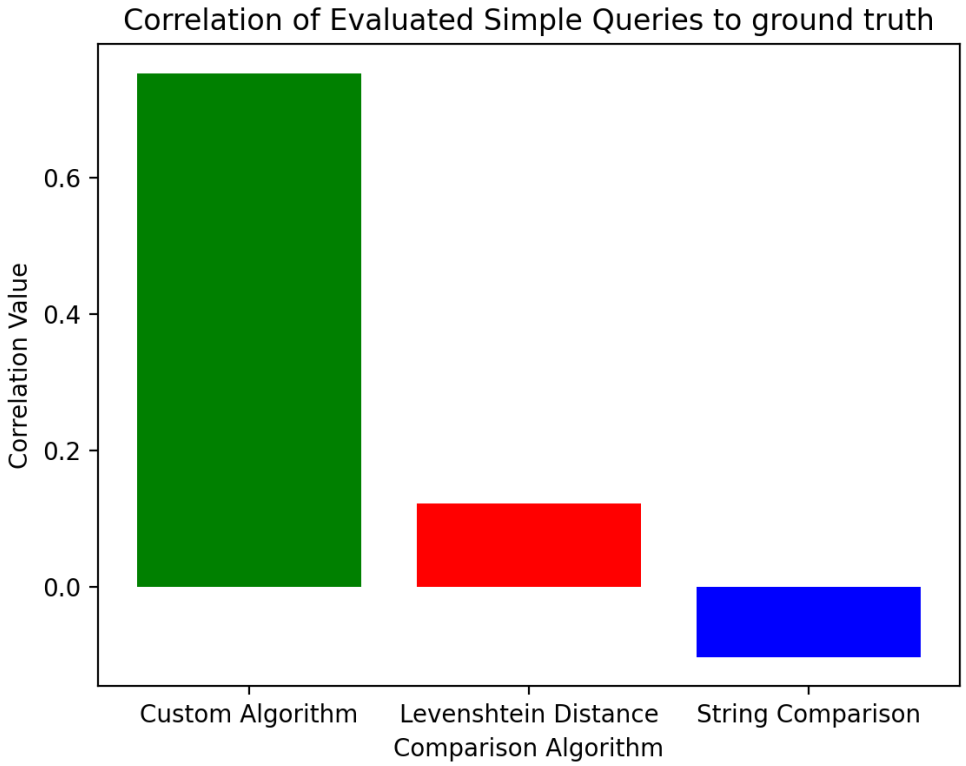

Figure 14: Correlation between true similar simple queries and query comparison algorithms results

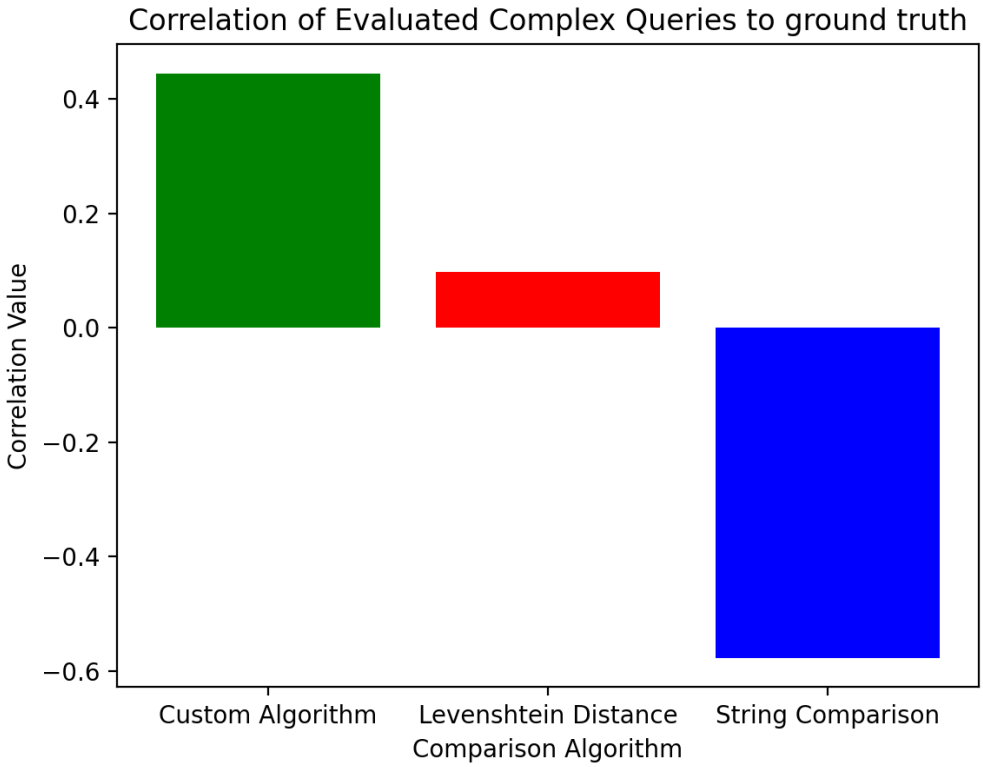

Figure 15: Correlation between true similar complex queries and query comparison algorithms results

We can clearly see a positive correlation when looking at our custom algorithm result and the Levenshtein distance algorithm, while the string comparison algorithm has a significant negative correlation coefficient, indicating its abysmal performance. Our custom algorithm performs particularly well when evaluating it using simple queries with a Pearson correlation of 0,7529, which indicates a significantly large positive relationship. When measuring the correlation in the complex queries evaluation, the custom algorithm still has the biggest correlation coefficient with a 0,2655 value. However, this is a much smaller and non-significant positive correlation.

Takeaway. Our custom algorithm performs the best out of the three comparison methods used in this experiment, as proven by the countless measurements mentioned above. It performs particularly well when comparing simple queries with one another. However, its performance is still not satisfactory when comparing complex queries that contain various syntactical elements, such as sub-queries,

unions, or different types of joins, as witnessed in our results.

5.4 Online Query Auditing Performance

In this section, we would like to evaluate the performance of the online query analysis, how much overhead it adds to the overall query execution process, and how robust PolicyLiner is to excessive traffic coming into the system. To evaluate the different performance facets of our online query auditing process, we used Locust to simulate the query requests to PolicyLiner [17].

5.4.1 Online Query Auditing Response Times

In this first experiment, we want to observe the average response times when we have only one user sending requests to PolicyLiner every 0,5 to 1 second. We ran this experiment for 240 seconds and executed 90 requests to PolicyLiner, where 38 of them contained complex queries, and 52 contained simple queries from the same set of queries that were used in the previous subsection experiments.

Table 14: Response Times of Online Query Auditing process

Response Times	**Simple**	**Complex**	**Total**
Median	150 ms	120 ms	140 ms
Average	156,55 ms	198,44 ms	174,24 ms
Max	304 ms	2507 ms	2507 ms
Min	36 ms	43 ms	36 ms

We can observe the response times in table 14. While the complex queries are on average slower, the simple queries have a higher median, which tells us that the complexity of the query is not correlated with the speed and response time of the online query auditing process. We also see a big outlier in the slowest response time for complex queries, being 2507 ms. Looking into figure 16 as well, we see that this request was the first one to be executed after PolicyLiner started running, so that is in all probability the reason why it is so much slower than the other queries. In Figure 16, we can also see that the response times become slower as time goes on. This can lead us to believe that either the added queries in the database start to make a difference in the performance of the process, or the machine on which we are running these experiments is starting to overheat and become slower.

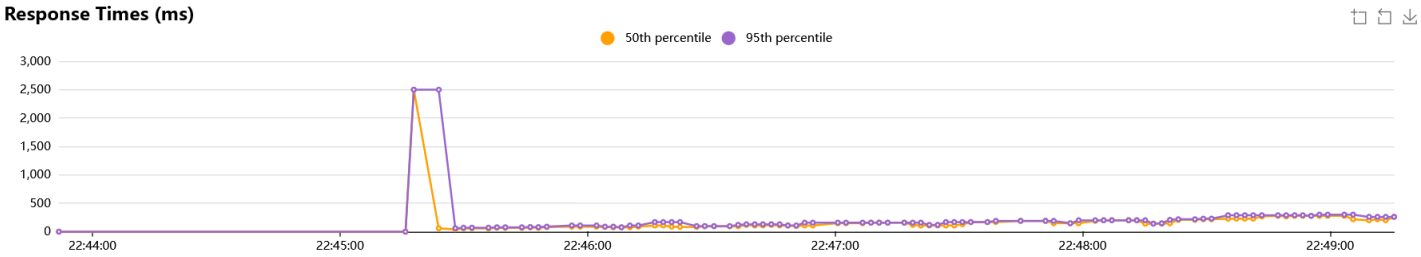

Figure 16: Response Times of Online Query Auditing Process

Takeaway. An average response time of 174,24 ms is something that is almost unnoticeable for the average user [32]:

- **0,1 second / 100 ms** is about the limit for having the user feel that the system is reacting instantaneously.
- **1,0 second / 1000 ms** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay.
- **10 seconds / 10000 ms** is about the limit for keeping the user's attention focused on the process.

Taking these thresholds in mind, an average response time of 174,24 ms still allows the system to take its time to return the query results to the user, when we take into account that a total response time of 1 second is more than acceptable by most users in most business cases. We would deem the performance of the online query auditing process sufficient for practical use in the real world.

5.4.2 Scalability

In this second experiment, we want to test if PolicyLiner can handle heavy traffic from many different users making multiple requests per second. We organized this experiment in the following way:

The Locust experiment was run for a total of five minutes. In the first minute, 50 users are spawned, each executing a request with a wait time of 1 to 2 seconds in between to simulate real-world usage. In the next two minutes, the number of users jumps to 100. In the fourth minute, we add another 50 users to bring the total to 150 users, and in the final minute, we add the final 50 users to complete the experiment with 200 concurrent users to PolicyLiner.

Table 15: Response Times of Online Query Auditing process with many concurrent users

Response Times	Simple	Complex	Total
Median	2400 ms	2400 ms	2400 ms
Average	2599,09 ms	2618,78 ms	2606,41 ms
Max	7587 ms	7053 ms	7587 ms
Min	4 ms	6 ms	4 ms

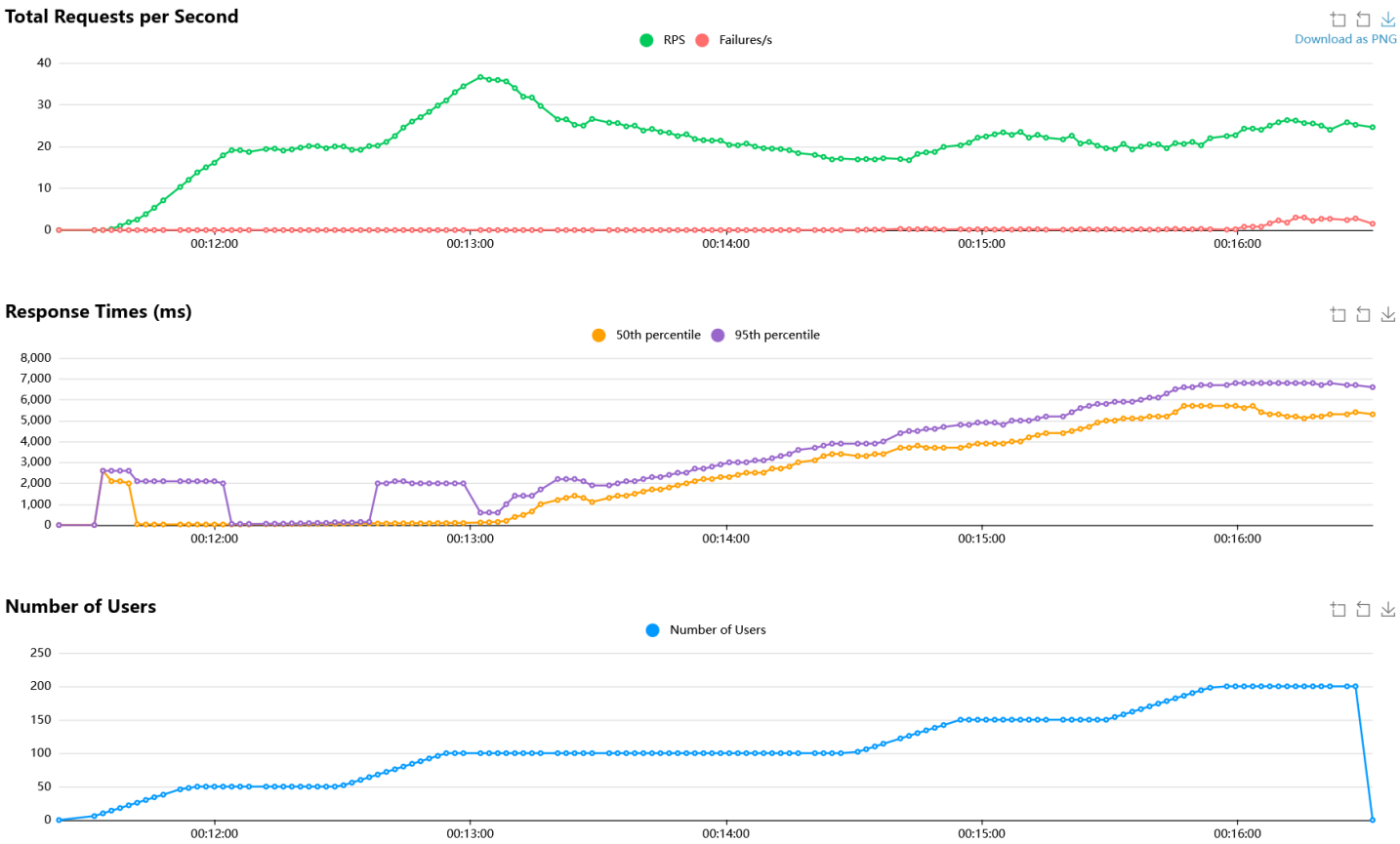

Figure 17: Scalability Experiment Graph Results

In this time, 6530 requests were completed, of which 2428 were requests containing complex queries and 4102 containing simple queries. A maximum of 36,6 requests per second was achieved in the second minute of the experiment with 100 concurrent users. We also saw some failed requests; in total, 82 requests or 1,25%

failed. The first failed request came in the fourth minute with 106 concurrent users and 17 requests per minute. The failed requests became more common in the final minute with 200 concurrent users. The maximum failure rate per second was 3, while the general requests per second frequency at the time was 26,2.

We can see the contrast in response time in table 15 with the table 14 of the previous experiment. It also becomes clear from Figure 17 when the change happens. Around the third minute of the experiment, when we have 100 concurrent users using PolicyLiner, and when thousands of disclosure queries had already been executed and stored in the PolicyLiner database, we see the upward trend in the response times of the user requests. Up until that point, outside of some outliers, which might have been caused by the computer throttling, the performance was really similar to our previous experiment. This leads us to believe that the more disclosure queries we store in the database and the more historical queries need to be compared with one another, the more the performance and response times degrade and slow down.

Takeaway. PolicyLiner can handle at least 200 concurrent users executing 20-36,6 requests per second at a fairly low request failure rate. On the one hand, considering the fact that this experiment is being done in a portable computer with a mainstream processor and other normal specifications, this leads us to believe PolicyLiner has acceptable performance even for practical real-world usage. On the other hand, the degradation and slowdown in performance after thousands of disclosure queries had already been analyzed and stored in the PolicyLiner database, makes a database query clean-up process necessary in widely used applications with high concurrent traffic in order for PolicyLiner to have a fast and efficient performance, and foremost to provide a usable and practical experience.

5.5 Offline Query Auditing Performance

In this section, we want to explore the performance of our offline query auditing process. We will evaluate this process across the different datasets mentioned in table 2 with a different number of queries being stored in the PolicyLiner database and inspected in each run. As thresholds for the different numbers of queries stored in the PolicyLiner database, we chose: 100 queries, 1000 queries, and 5000 queries. The disclosure policies that were used in these experiments were the same across the different datasets and the number of disclosure queries. They are defined in our PolicyLiner Github repository: *PolicyLiner disclosure policies*

5.5.1 100-Patients Dataset

As displayed in the title of this subsection, we start our experiment with the smallest dataset shown in table 2. The disclosure policies, from which the disclosure queries aim to acquire the data, have their sizes displayed in table 16.

Table 16: 100-Patients Disclosure Policy Sizes

Disclosure Policy	Size
admissions_diagnosis_policy	1602 rows
labs_policy	111.483 rows
patients_admissions_diagnosis_policy	1602 rows
patients_admissions_diagnosis_policy944	1602 rows
patients_admissions_policy	372 rows
patients_policy	100 rows

100 disclosure queries

We run our experiment first with the smallest size of queries and observe that the whole process takes 10,471 seconds to complete. A more detailed look into each individual query evaluation is given in Figure 18.

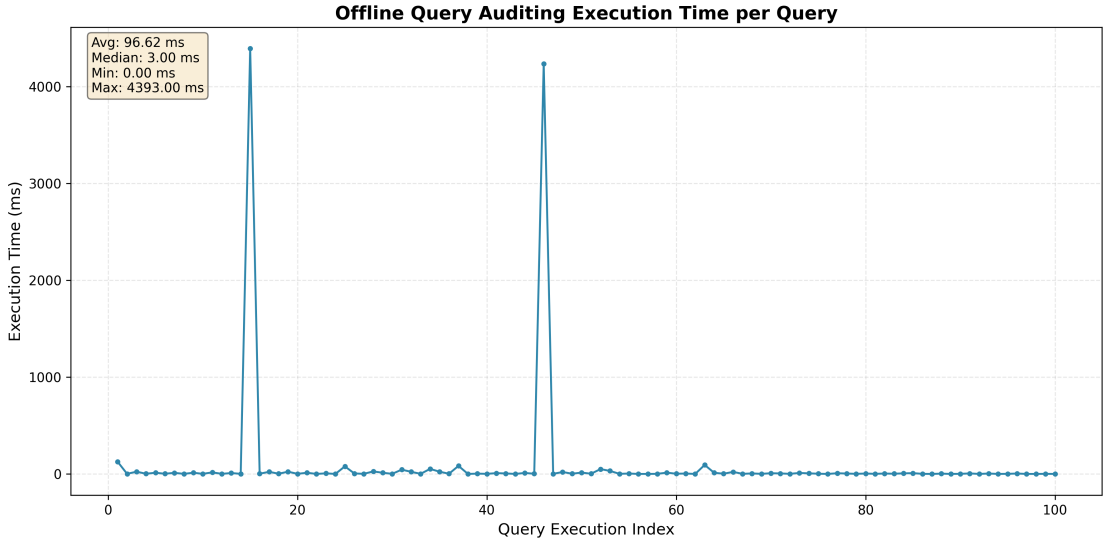

Figure 18: Offline Query Auditing Performance per Query in 100-Patients dataset with 100 queries in total

We can clearly see two outliers in the whole experiment. Upon taking a closer look as to what the reason is for these outliers, we observe that they both are using

the *labs_policy* view and because it is much bigger than the other policies, we see these big differences in execution times. These two disclosure queries make up more than 82% of the whole execution time of the experiment, with each of them taking 4393 ms and 4236 ms, respectively, to complete. All the queries took on average 96,62 ms to complete, with the minimum taking less than 0,5 ms. We also see the effect that the two outliers have on the average value when we calculate the median, which is just 3 ms.

1000 disclosure queries

We run our second experiment with 1000 disclosure queries and observe that the whole process took 44,835 seconds. A more detailed look into each individual query evaluation is given in Figure 19.

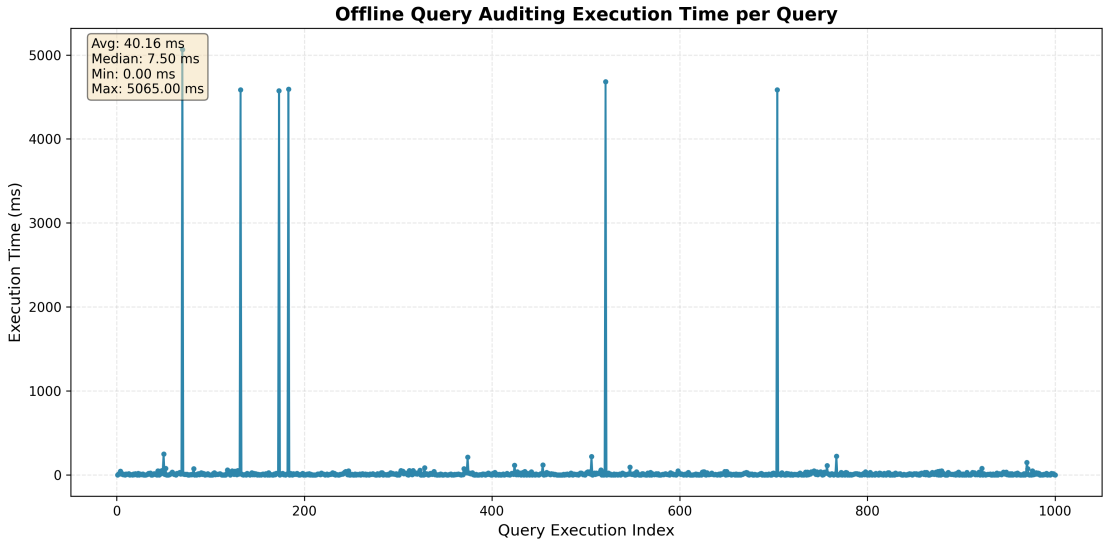

Figure 19: Offline Query Auditing Performance per Query in 100-Patients dataset with 1000 queries in total

We can clearly see some outliers in these experiments, just as in the previous one, and the reason for them is the same as for the previous experiment, the queries are using the *labs_policy* view, and because of its size, the execution times become bigger exponentially in comparison with the other query evaluations. We see that the maximum time of 5065 ms is somewhat bigger than the previous experiment with 100 queries, but still comparable and within the margin of error. The minimum, just as before, is less than 0,5 ms. The average is a lot lower this time, with 40,16 ms. This is likely due to the fact that there are fewer outliers in proportion to the whole query size of 1000, which skews the average upwards. This

is also recognized in the Median value of 7,5 ms, which is more than double the value of the previous experiment. Thus, we see that the number of queries stored in the database makes a difference in the evaluation speed of individual queries, as each user has executed more queries, and that means that each individual query must be compared with more queries on average than in the previous experiment. The maximum queries executed by one user in this experiment were 12, while in the previous experiment, it was just 2.

5000 disclosure queries

We run our third experiment with 5000 disclosure queries stored in the PolicyLiner database and observe that the whole process takes 1239,433 seconds or about 20 minutes. A more detailed look into each individual query evaluation is given in Figure 20.

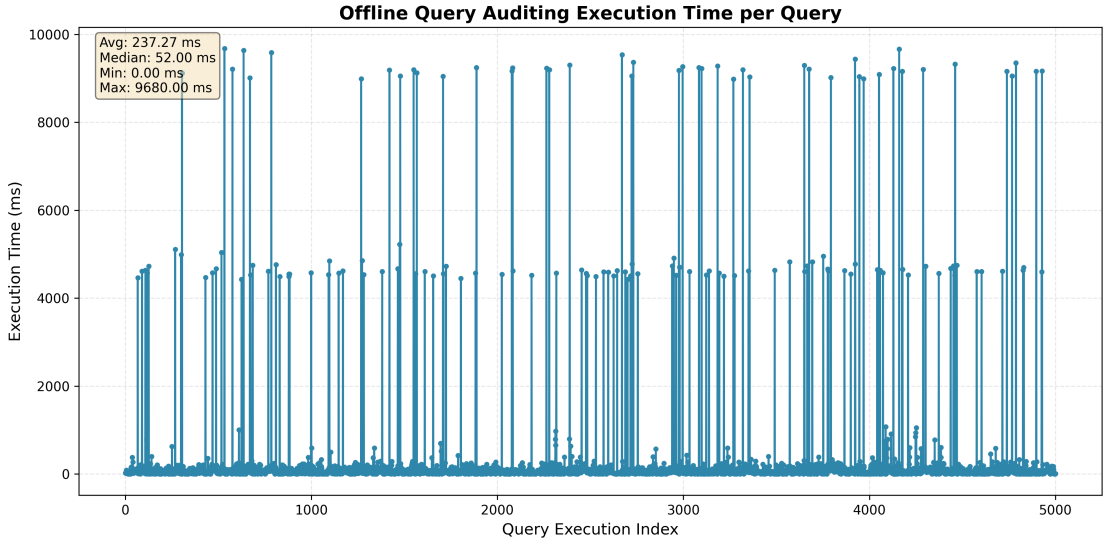

Figure 20: Offline Query Auditing Performance per Query in 100-Patients dataset with 5000 queries in total

We see that in this experiment, as with the other two, there are outliers that coincide with queries that are using the *labs_policy* view. In this third experiment, we see that the average execution time balloons up to 237,27 ms and the median to 52 ms. The maximum is also almost double the previous value, with 9680 ms. This is partly due to the fact that there are a lot more queries using the *labs_policy* view than before, and partly because each user has, of course, executed a lot more queries individually, with the maximum being 48. This makes the evaluation slower for each query and raises the general execution times.

Takeaway The speed of the offline query auditing process is strongly dependent on the disclosure policy, view sizes, and the quantity of queries each user executes individually. In our opinion, this should not be a bottleneck to using PolicyLiner in practice, as the offline query auditing process does not interfere with the other requests that might be made at the same time, as it is running on a different virtual thread.

5.5.2 10000-Patients Dataset

We continue our experiments with the 10000-Patients dataset. The disclosure policies, from which the disclosure queries aim to acquire the data, have their sizes displayed in table 17. Because of our transaction timeout limit mentioned at the top of chapter 5, we excluded queries that use the *labs_policy* view, as its big size triggered the timeout and threw an exception when used.

Table 17: 10000-Patients Disclosure Policy Sizes

Disclosure Policy	Size
admissions_diagnosis_policy	154.243 rows
labs_policy	10.726.505 rows
patients_admissions_diagnosis_policy	154.243 rows
patients_admissions_diagnosis_policy944	154.243 rows
patients_admissions_policy	36.143 rows
patients_policy	10.000 rows

100 disclosure queries

As with the previous dataset, we start with 100 disclosure queries in the 10000-Patients dataset as well. The whole process took 9120 ms to complete. In this experiment, we see that the average query was evaluated in 89,4 ms with the median being 5 ms. We see some outliers in the data that take more than 1000 ms to complete, with the slowest query evaluation being 1914 ms. We do not have a clear outlook into why these outlier evaluation times happened, but it could be a combination of different factors, such as the computer lagging or the same user having executed a sizable number of queries. Another interesting thing in this experiment comes from its duration as a whole being about a second shorter than the same experiment on the 100-patients dataset. Although this looks counterintuitive at first sight, we can find the reason for this happening fairly easily, and it's the fact that the *lab_policy* view queries were excluded from these experiments.

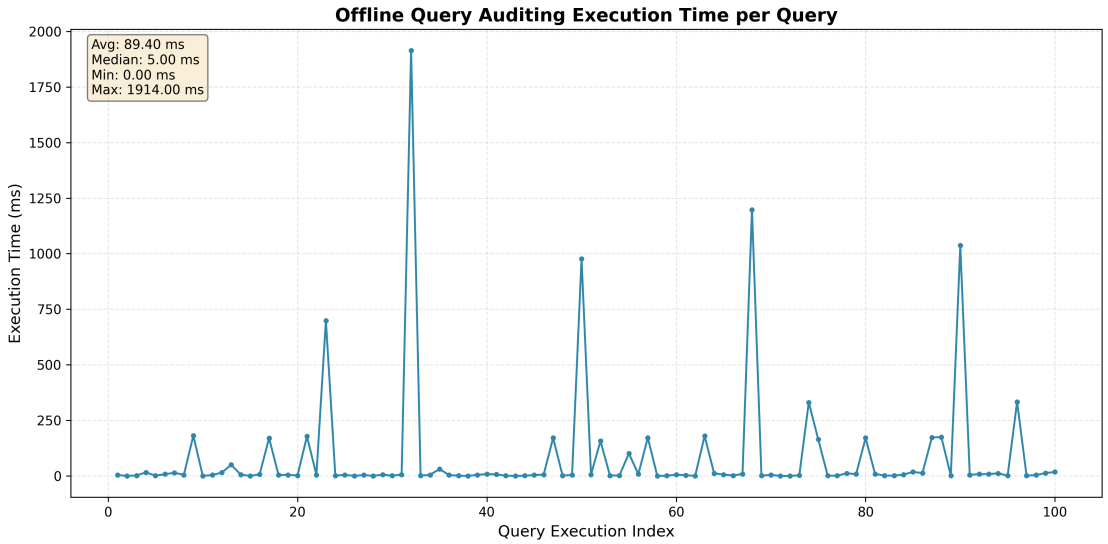

Figure 21: Offline Query Auditing Performance per Query in 10000-Patients dataset with 100 queries in total

1000 disclosure queries

We continue with the sample size of 1000 disclosure queries. It took 1.220.583 ms or about 20 minutes as a whole. In this experiment, we also see a lot of outliers. The interesting thing in this case is that the outliers all come from one user. We believe that the reason for this happening is that this user executed a lot of similar queries that used the same tables and the same or similar columns, so most of his queries had to be compared to one another, thus making the evaluation time balloon as it has. The other users, on the other hand, have executed various queries from different views, so not every one of the queries needs to be compared with one another, and as such, the evaluation time shortens.

In this experiment, we see an average query evaluation time of 1215,05 ms and a median evaluation time of 33 ms. The maximum single query evaluation time is 110.328 ms, and the minimum is less than 0,5 ms. A fun fact about this experiment is that when we calculate the sum of the evaluation time of the queries from the one user who created all the outliers in Figure 22, we find out that it is 498.957 ms. That means that more than 40% of the whole evaluation time is taken from this user, despite him executing only 17 out of the 1000 evaluated queries in this process.

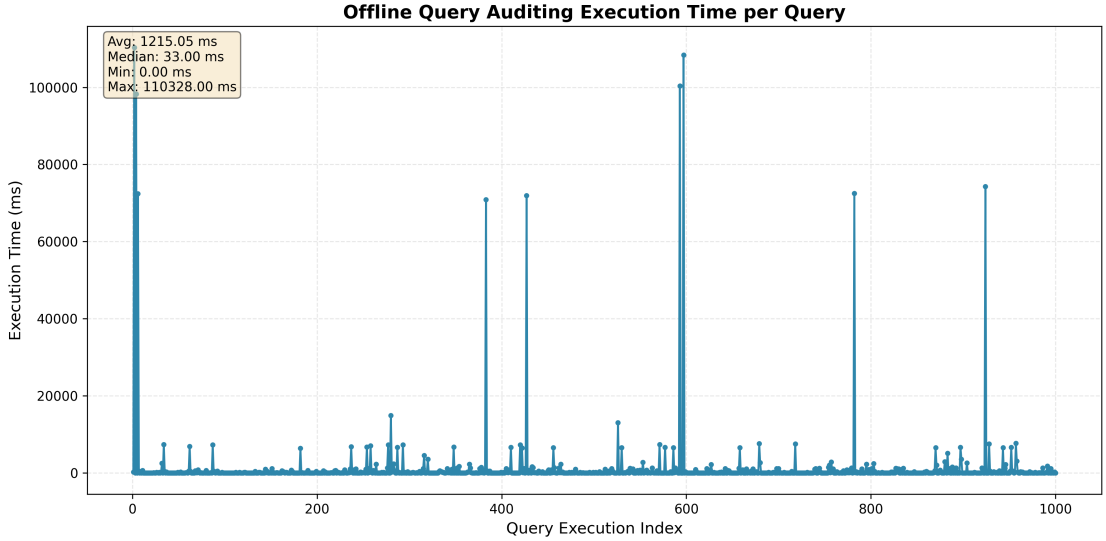

Figure 22: Offline Query Auditing Performance per Query in 10000-Patients dataset with 1000 queries in total

5000 disclosure queries

Lastly comes the sample size of 5000 disclosure queries. This experiment, unfortunately, triggers the transaction timeout when run. Thus, it could not be completed. The cause of this timeout is, in all likelihood, the fact that the individual users have executed a larger number of queries. In comparison with the previous experiment, where the user with the most queries had executed 23 of them, in this experiment, the most active user had executed 56 queries.

Takeaway. We see that while the database size plays a big role in the performance of our offline query auditing process, the number, type, and distribution of queries between the different users arguably play an even bigger role in the individual query evaluation time as well as the ability to complete the whole process with no exceptions.

5.5.3 100000-Patients Dataset

We now want to complete the same experiments as above with the biggest dataset in our lineup. The disclosure policies, from which the disclosure queries acquire their data, have their sizes displayed in table 18. Once again, as with the previous dataset 10000-Patients, we are excluding the queries that receive their data from *labs_policy* view because of its enormous size.

Table 18: 100000-Patients Disclosure Policy Sizes

Disclosure Policy	Size
admissions_diagnosis_policy	1.540.584 rows
labs_policy	107.535.387 rows
patients_admissions_diagnosis_policy	1.540.584 rows
patients_admissions_diagnosis_policy944	1.540.584 rows
patients_admissions_policy	361.759 rows
patients_policy	100.000 rows

100 disclosure queries

We start the experiments on the biggest dataset in the same way as the others, with the smallest sample of disclosure queries: 100. The whole process took a total of 214.958 ms or about 3,5 minutes to complete. This is significantly slower than the other two datasets with the 100 sample size. The individual query evaluation was completed on average in 2149,58 ms, and the median was 4,5 ms. The reasons for the big discrepancy are mostly the two biggest outliers, which took 66.978 ms and 65.610 ms, respectively, and as such, together make up more than half of the whole run time of the offline query auditing process. The reason why these outliers exist is not completely clear; however, we believe that the queries being evaluated are complex and, as such, prolong the process.

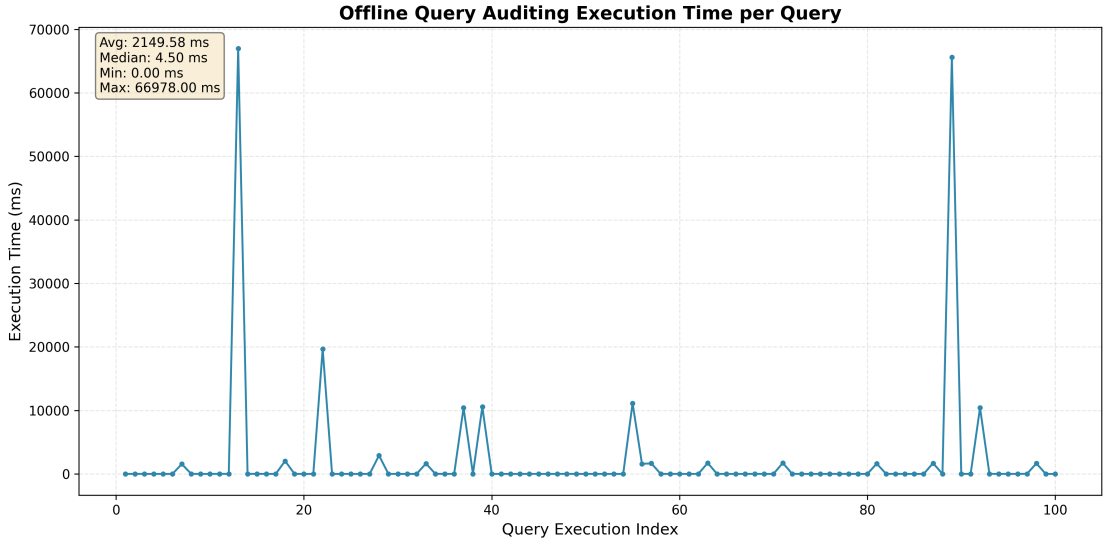

Figure 23: Offline Query Auditing Performance per Query in 100000-Patients dataset with 100 queries in total

1000 disclosure queries

We now continue with the sample size of 1000 disclosure queries. This whole process took 7.623.488 ms or more than 2 hours to complete. Unfortunately, the individual statistics from this long process could not be gathered, and as such, no graph can be created. However, from the overall length of the process, we can see that executing the offline query auditing process in datasets with millions of data points and thousands of executed disclosure queries proves hard to do.

5000 disclosure queries

Lastly comes the sample size of 5000 disclosure queries. Unfortunately, as with the previous, smaller dataset, this experiment triggers the transaction timeout and, as such, cannot be completed.

Takeaway. Running our offline query auditing process in large datasets is cumbersome, even though it runs in an unblocking virtual thread, and an optimization of the process or the number of disclosure queries evaluated per process can be an interesting topic for future research.

5.6 Offline Policy Auditing Performance

In this section, we aim to evaluate the performance and efficiency of the offline policy auditing process. We want to achieve this by testing the offline policy auditing process across various disclosure policies and the different datasets mentioned in table 2. We will run the offline policy auditing process multiple times and calculate the average time it takes for it to be completed. We will also research the individual privacy metrics that we implemented and evaluate their speed and impact on the performance of the whole process. The graphs in this section will contain the policies as the x-axis, but since the policy names are somewhat long to use in the plots unshortened, we shortened them in the following way:

- **adp**: *admissions_diagnosis_policy*
- **lp**: *labs_policy*
- **padp**: *patients_admissions_diagnosis_policy*
- **padp944**: *patients_admissions_diagnosis_policy944*
- **pap**: *patients_admissions_policy*
- **pp**: *patients_policy*

The sizes of each of the policies in each dataset are displayed in tables 16, 17, and 18.

Firstly, we want to show the overall performance of the offline policy auditing process, and then we will continue with the individual performance of each privacy metric we have implemented in the offline policy auditing process.

In figure 24, we can see the overall performance of the offline policy auditing process across our three different datasets, as well as the evaluation performance of the individual disclosure policies. The policy auditing process in the 100-Patients dataset took less than 6 seconds as a whole, with 5 seconds being allocated only towards the disclosure policy with the biggest size: *labs_policy*. In the next two datasets, *labs_policy* was excluded from the experiments because it triggered the transaction timeout limit of PolicyLiner. In spite of this fact, we see an exponential increase in the evaluation time of the 10000-Patients dataset policies. It took a total of 198.979 ms or about 3 minutes to complete the offline policy auditing process. The interesting thing in the next dataset is that the time does not increase so dramatically as from the smallest dataset to the 10000-Patients dataset. The offline policy auditing process took on average 228.591,7 ms or less than 4 minutes in the 100000-Patients. This is mostly due to the fact that the *t*-closeness metric could not be measured on the *patients_admissions_policy* and *patients_policy* in the 100000-Patients dataset, as it reached our transaction timeout limit.

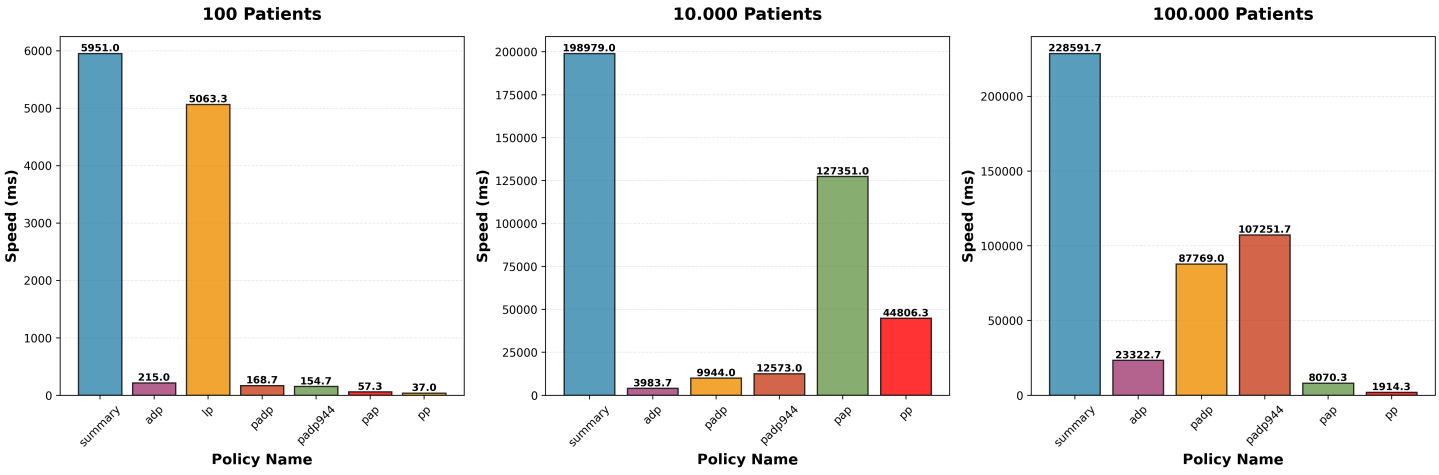

Figure 24: Overall Offline Policy Auditing Performance Comparison Across Datasets

This is also proved by the fact that the other policies that are not affected by this change have significantly higher evaluation times than the 10000-Patients dataset evaluation, from about six to nine times higher. The average evaluation time for the 100-Patients dataset is 1663,86 ms, for the 10000-Patients dataset is 66272,83

ms, and for the 100000-Patients dataset is 76153,28 ms. We will see the exact influence of t-closeness and the other privacy metrics on the overall evaluation time in the next subsections.

5.6.1 δ-Presence Performance

In this subsection, we will take a closer look at the δ-Presence privacy metric evaluation times. In figure 25 we see average execution times of 134,78 ms on the 100-Patients dataset, 577,33 ms in the 10000-Patients dataset, and 5499,27 ms in the 100000-Patients dataset. We see very big increases in the duration of the evaluation of each of the disclosure policies, moving from one dataset to the other. Another consistent fact we have observed is that the average duration of the δ-Presence evaluation is less than one-tenth of the length of the complete offline policy auditing process, thus making this privacy metric evaluation one of our more efficient evaluations. Additionally, we see no surprises in the evaluation times of the individual disclosure policies, with the bigger-sized policies taking longer to evaluate than the smaller-sized ones.

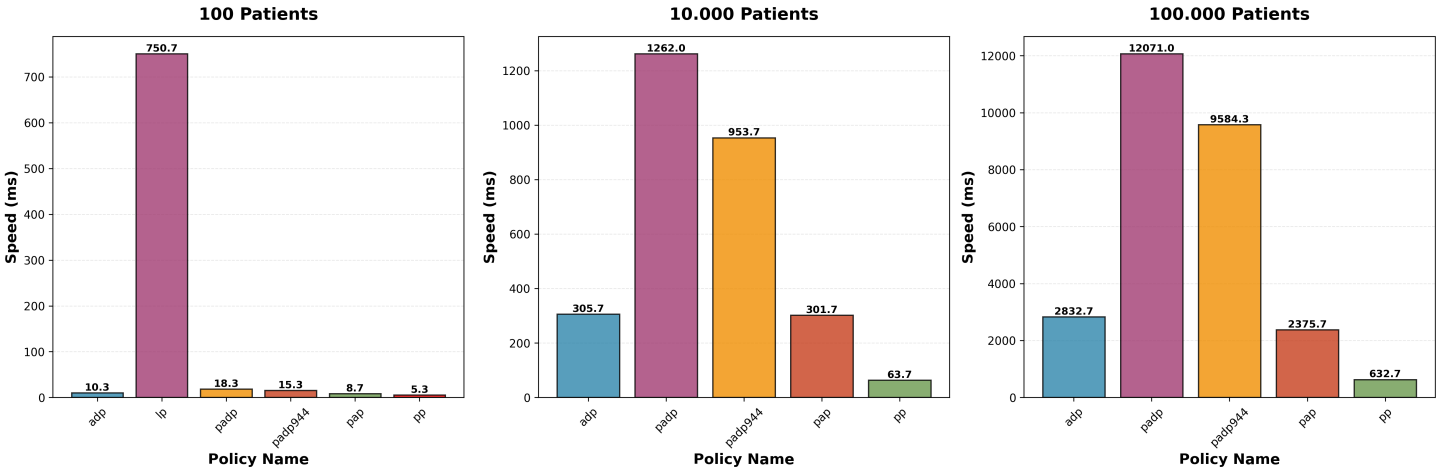

Figure 25: δ-Presence Performance Comparison Across Datasets

5.6.2 Sample Uniqueness Ratio Performance

In this subsection, we will talk about the performance of the sample uniqueness ratio metric, depicted in Figure 26. The average execution lengths for this metric are:

- 139,33 ms for the 100-Patients dataset

- 539,80 ms for the 10000-Patients dataset
- 5128 ms for the 100000-Patients dataset

This is comparable to the δ-Presence privacy metric. We also see no surprises with this privacy metric; the bigger the size of the disclosure policy, the longer the evaluation time. We see a meaningful difference in evaluation length between *patients_admissions_diagnosis_policy* and *patients_admissions_diagnosis_policy944* even though they contain the same number of entries in their respective views. This is most likely due to the fact that the *patients_admissions_diagnosis_policy* view contains two more columns than the *patients_admissions_diagnosis_policy944* view, which increases its data volume and, as such, the evaluation length.

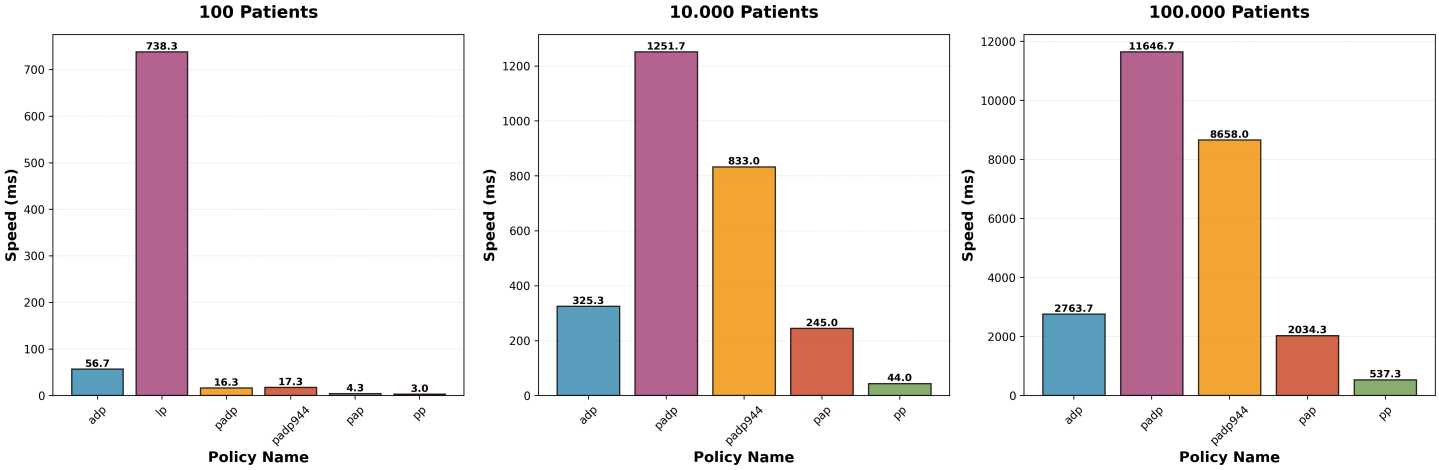

Figure 26: Sample Uniqueness Ratio Performance Comparison Across Datasets

5.6.3 Population Uniqueness Estimation Performance

In this subsection, we will talk about the performance of the population uniqueness estimation metric, displayed in Figure 27. The average execution lengths for this metric are:

- 244,11 ms for the 100-Patients dataset
- 905,67 ms for the 10000-Patients dataset
- 8699,47 ms for the 100000-Patients dataset

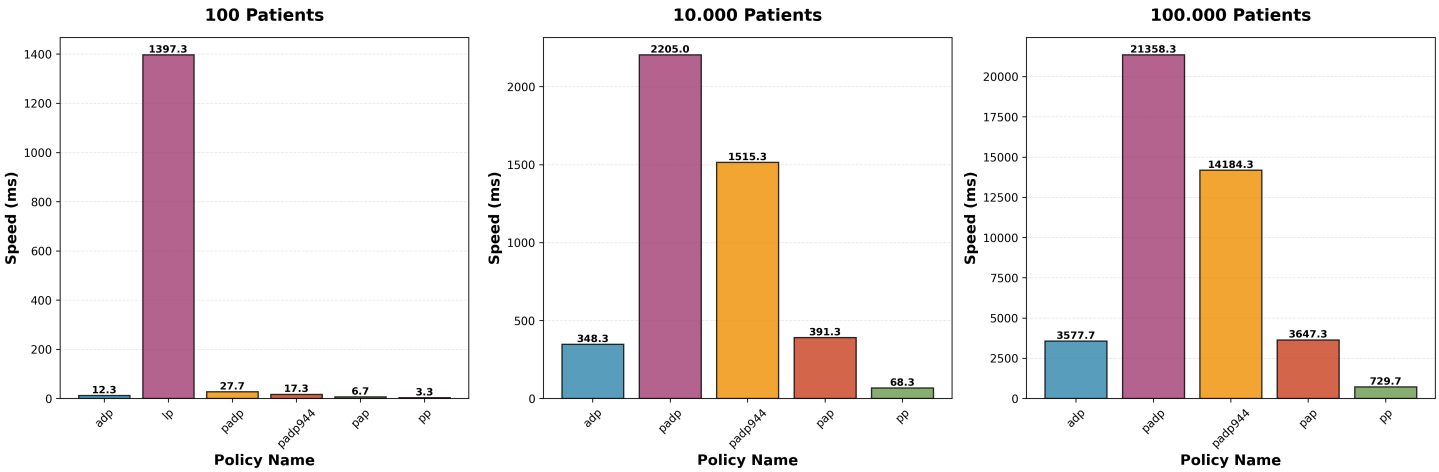

Figure 27: Population Uniqueness Estimation Performance Comparison Across Datasets

It is clearly slower than the previous two metrics, but it still continues the same patterns regarding the duration of each individual policy as well as the exponential duration increase between the different datasets from the smallest to the biggest.

5.6.4 T-Closeness Performance

Lastly, we evaluate the t-Closeness privacy metric performance as displayed in Figure 28.

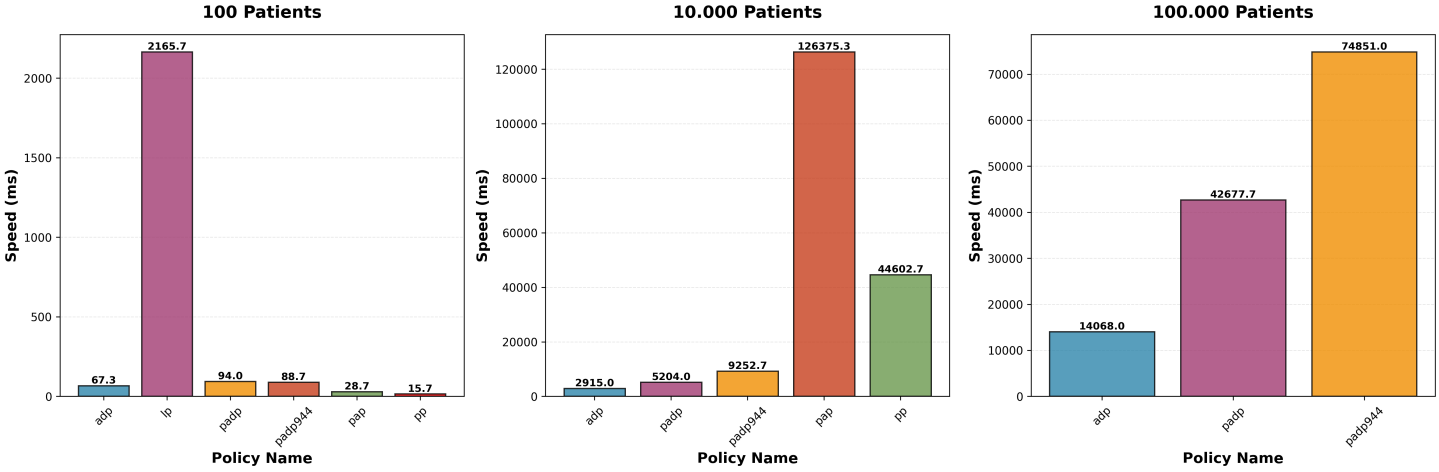

Figure 28: t-Closeness Performance Comparison Across Datasets

The average execution lengths for this metric are:

- 410 ms for the 100-Patients dataset
- 37669,93 ms for the 10000-Patients dataset
- 43865,56 ms for the 100000-Patients dataset

This is by far our slowest privacy metric in the offline policy auditing process, and it constitutes more than half the overall offline policy auditing process length. This is likely the case because the t-Closeness metric deals with sensitive attributes in addition to the equivalence classes and, as such, has to run multiple queries in the disclosure policy views to receive all the needed information in the required distribution. Another interesting finding we see in figure 28 is that it does not follow the usual pattern of the other metrics, where the bigger disclosure policy had the longest evaluation duration. In fact, the policy with the longest evaluation length (excluding *lab_policy*, which unfortunately triggers the timeout in the bigger datasets) is *patients_admissions_policy*, which has the second smallest view size of the tested disclosure policies, and the second-longest evaluation duration is from *patients_policy*, which has the smallest view size of all. That is also the reason these two disclosure policies trigger the transaction timeout when evaluating them in the biggest dataset. Our hypothesis about why this is the case is that these two disclosure policies contain a lot of smaller equivalence classes, and as the distribution of the sensitive attributes must be calculated for each and every one of them, the duration of the t-Closeness evaluation becomes a lot larger in these policies in comparison to the disclosure policies with a somewhat bigger view size.

Takeaway. The performance of the offline policy auditing process is influenced by many different factors. The most important one is the dataset and disclosure policy view sizes. However, the equivalence class sizes as well as sensitive attribute distribution between the equivalence classes and the whole dataset play a significant role in the performance of the individual privacy metrics measured during this auditing process, in particular the t-Closeness privacy metric.

6 Related Work

In this chapter, we would like to further discuss papers and works that, while not needed as a background to our thesis, still relate to it.

6.1 Privacy Auditing / Benchmarking

In our research regarding privacy auditing, we found that a large number of research papers dealt with the auditing of privacy in the machine learning field, and especially regarding differential privacy techniques.

Steinke et al. [46] showcase an auditing procedure that can audit the differential privacy guarantees of an algorithm with a single training run in differentially private machine learning systems.

Tramer et al. [49] show that even though differential privacy can provide provable privacy guarantees, that does not prove the absence of errors. They use auditing to find flaws in differentially private schemes and showcase their findings in an open source implementation of a differentially private deep learning algorithm, and find with 99.99999999% confidence that the implementation does not satisfy the claimed differential privacy guarantee[49].

Chadha et al. [5] propose a framework for auditing private prediction where they instantiate adversaries with varying poisoning and query capabilities.

Chang et al. [6] study privacy auditing for federated learning. They design a novel, efficient membership inference attack to audit privacy risks in federated learning. Their solution does not need to train any additional complicated learning models or perform costly computations. It can be integrated by the parties seamlessly, without disrupting the federated learning process[6].

Nasr et al. [28] design an improved auditing scheme that yields tight privacy estimates for natural (not adversarially crafted) datasets. Their auditing scheme requires only two training runs (instead of thousands) to produce tight privacy estimates, by adapting recent advances in tight composition theorems for differential privacy[28].

In addition to these kinds of papers, there is, of course, literature that deals with privacy auditing and benchmarking outside of machine learning and differential privacy.

Shastri et al. [44] design and implement an open-source benchmark called *GDPRbench* that consists of workloads and metrics needed to understand and assess personal-data processing database systems. They analyze GDPR from a systems perspective, translating its legal articles into a set of capabilities and characteristics that compliant systems must support [44]. They pick the most strict interpretation of the law in order to find out the worst-case performance costs of GDPR. They make three key observations during their analysis:

1. They find that each personal data item is associated with up to seven metadata properties that govern its behavior. This is significant as it severely impacts database operations to manipulate or view these pieces of data and leads to something called *metadata explosion*.

2. GDPR's goal of data protection by design and by default conflicts with the traditional system design goals of optimizing for performance, cost, and reliability.

3. They identify that GDPR allows new forms of interactions with datastores called *GDPRqueries*, which they organize in their *GDPRbench* benchmark.

To conclude, they demonstrate that compliance incurs large overheads and that scalability is poor when large volumes of personal data and metadata are involved. Their work provides one of the first system-oriented analyses of GDPR compliance costs and lays the groundwork for benchmarks and system design to support GDPR-aware data management.

7 Conclusion

This thesis set out to address a central problem in modern data-driven environments: how to ensure that disclosure-compliant policies on sensitive datasets remain up-to-date and effective in environments with rapidly evolving and dynamic data. Current systems focus either on data integrity and consistency, or they lack the mechanisms to continuously monitor, evaluate, and update disclosure policies in response to data alterations. Therefore, we proposed PolicyLiner, an open-source database auditing tool that works closely with Mascara to ensure that disclosure policies remain compliant through consistent data changes in dynamic environments and that disclosure-compliant queries are evaluated based on users' previous query history. We created three independent processes to achieve this: *Online Query Auditing*, *Offline Query Auditing*, and *Offline Policy Auditing*. These processes audit and identify vulnerabilities to four specific privacy attacks and alert the data officer to any issues they observe. In this way, disclosure policies are kept compliant, and disclosure queries do not provide sensitive information to malicious actors.

Based on our exhaustive experiments, PolicyLiner can effectively recognize privacy vulnerabilities arising from changing data in the database using our *Offline Policy Auditing* process and does so at a reasonable speed, even though the process runs on its own unblocking thread, and the process duration is not of first importance. In addition, we evaluated our custom-implemented algorithm for query comparison that takes place during our *Online Query Auditing* process and found that while it performs very well when comparing simpler queries with one another, it still lacks the complexity and expertise to differentiate similar and similarly looking complex queries. Furthermore, we tested the overall performance and speed of the *Online Query Auditing* process. We found that as long as the number of historical disclosure queries stored in the database is not massive, the process can complete in an almost unrecognizable time by the user of about 170 ms. We also evaluated the scalability and robustness of the *Online Query Auditing* process and achieved a maximum of 200 concurrent users and 36.6 requests per second, with a small failure rate of just 1%, while executing over 6000 query requests on my personal portable computer.

7.1 Future Work

While PolicyLiner provides a comprehensive foundation for database privacy auditing, several promising directions remain for future research and development:

Improved Population Uniqueness Estimation. We implemented the Population Uniqueness Estimation metric according to Dankar et al [8]. Unfortunately, this privacy metric could not be thoroughly tested, and we cannot guarantee its current correctness and effectiveness in all cases. This should be addressed in future work.

Custom-Implemented Query Comparison Algorithm. Our custom-implemented query comparison algorithm performs very well when comparing simple queries. Its performance degrades significantly when comparing more complex queries. An interesting future work direction would be to extend our implementation to support more complex queries that include less common query parts, such as *unions*, sub queries, and *group by* clauses.

Offline Auditing Processes Performance. In our current machine setup, we could not test the performance of the offline query auditing and offline policy auditing processes when using big datasets, as the transactions would reach the timeout limit. We should test these scenarios on more powerful machines and, if performance is insufficient, explore faster, more optimized solutions.

Automatic policy repair and recommendation. Currently, PolicyLiner sends alerts to data officers to make them aware of the vulnerabilities the datasets they manage might have. A natural extension of this capability would be an automatic policy-repair feature that suggests safer masking functions for the data officer to implement, leveraging PolicyLiner's current auditing functionalities.

7.2 AI Usage

During the construction of this thesis, ChatGPT (available at `https://chatgpt.com/`) was partly utilized for literature research and to support the creation of the abstract. Claude AI (available at `https://claude.ai`) was used in addition to assist with the creation of the plots in Chapter 5, the masking functions stored in our Github repository: *PolicyLiner masking functions* and the creation of random similar and similarly looking queries to use in the evaluation of our custom query comparison algorithm. Grammarly (available at `https://app.grammarly.com/`) was used to assist with fixing grammatical errors in this writing and as a spell check mechanism.

List of Figures

List of Tables

List of Algorithms

List of Code Fragments

Bibliography

[1] What is data masking? techniques, types and best practices, https://www.techtarget.com/searchsecurity/definition/data-masking

[2] Top data anonymization tools for 2025 (2025), https://www.k2view.com/blog/data-anonymization-tools/, accessed: 2025-11-07

[3] Almasi, M.M., Siddiqui, T.R., Mohammed, N., Hemmati, H.: The risk-utility tradeoff for data privacy models. In: 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5. IEEE (2016)

[4] Anwar, M.R., Panjaitan, R., Supriati, R.: Implementation of database auditing by synchronization dbms. International Journal of Cyber and IT Service Management (IJCITSM) 1(2), 197–205 (2021)

[5] Chadha, K., Jagielski, M., Papernot, N., Choquette-Choo, C.A., Nasr, M.: Auditing private prediction. In: Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., Berkenkamp, F. (eds.) Proceedings of the 41st International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 235, pp. 6066–6092. PMLR (21–27 Jul 2024), https://proceedings.mlr.press/v235/chadha24a.html

[6] Chang, H., Edwards, B., Paul, A.S., Shokri, R.: Efficient privacy auditing in federated learning. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 307–323. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/chang

[7] Chen, G., Keller-McNulty, S.: Estimation of identification disclosure risk in microdata. Journal of Official Statistics 14(1), 79 (1998)

[8] Dankar, F.K., El Emam, K., Neisa, A., Roffey, T.: Estimating the re-identification risk of clinical data sets. BMC medical informatics and decision making 12(1), 66 (2012)

[9] Desfontaines, D.: -presence, for when being in the dataset is sensitive. https://desfontain.es/blog/delta-presence.html (04 2018), ted is writing things (personal blog)

[10] Dosselmann, R., Sadeqi, M., Hamilton, H.J.: A tutorial on computing t-closeness (2019), https://arxiv.org/abs/1911.11212

[11] El Emam, K., Dankar, F.K.: Protecting privacy using k-anonymity. Journal of the American Medical Informatics Association 15(5), 627–637 (Sep 2008)

[12] El Emam, K., Dankar, F.K., Issa, R., Jonker, E., Amyot, D., Cogo, E., Corriveau, J.P., Walker, M., Chowdhury, S., Vaillancourt, R., Roffey, T., Bottomley, J.: A globally optimal k-anonymity method for the de-identification of health data. Journal of the American Medical Informatics Association 16(5), 670–682 (Sep 2009)

[13] European Parliament, Council of the European Union: Regulation (EU) 2016/679 of the European Parliament and of the Council, https://data.europa.eu/eli/reg/2016/679/oj

[14] Gale, W.A., Sampson, G.: Good-turing frequency estimation without tears. Journal of quantitative linguistics 2(3), 217–237 (1995)

[15] Google: Angular: The framework for building scalable web apps with confidence

[16] Groomer, S.M., Murthy, U.S.: Continuous auditing of database applications: An embedded audit module approach1. In: Continuous Auditing: Theory and Application. Emerald Publishing Limited (03 2018), https://doi.org/10.1108/978-1-78743-413-420181005

[17] Heyman, J., Holmberg, L.: Locust: An open source load testing tool

[18] Hiremath, S., Kunte, S.: A novel data auditing approach to achieve data privacy and data integrity in cloud computing. In: 2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT). pp. 306–310 (2017)

[19] Ingle, M.V., Deshmukh, M.: Overview of database query auditing techniques (2014)

[20] Kartoun, U.: A methodology to generate virtual patient repositories (2016), https://arxiv.org/abs/1608.00570

[21] Levenshtein, V.: I.(1966) binary codes capable of correcting deletions, insertions and reversals. In: Soviet Physics Doklady. vol. 10, p. 707

[22] Li, N., Li, T., Venkatasubramanian, S.: t-closeness: Privacy beyond k-anonymity and l-diversity. In: 2007 IEEE 23rd international conference on data engineering. pp. 106–115. IEEE (2006)

[23] Liu, L., Huang, Q.: A framework for database auditing. In: 2009 Fourth International Conference on Computer Sciences and Convergence Information Technology. pp. 982–986 (2009)

[24] Liu, Y.: Build an audit framework for data privacy protection in cloud environment. Procedia Computer Science 247, 166–175 (2024), https://www.sciencedirect.com/science/article/pii/S1877050924028205, the 11th International Conference on Applications and Techniques in Cyber Intelligence

[25] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: l-diversity: Privacy beyond k-anonymity. Acm transactions on knowledge discovery from data (tkdd) 1(1), 3–es (2007)

[26] Motwani, R., Nabar, S.U., Thomas, D.: Auditing sql queries. In: 2008 IEEE 24th International Conference on Data Engineering. pp. 287–296 (2008)

[27] Murthy, S., Abu Bakar, A., Abdul Rahim, F., Ramli, R.: A comparative study of data anonymization techniques. In: 2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS). pp. 306–309 (2019)

[28] Nasr, M., Hayes, J., Steinke, T., Balle, B., Tramèr, F., Jagielski, M., Carlini, N., Terzis, A.: Tight auditing of differentially private machine learning. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 1631–1648. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/nasr

[29] Near, J.P., Darais, D., Lefkovitz, N., Howarth, G.S.: Guidelines for evaluating differential privacy guarantees. NIST Special Publication 800-226, National Institute of Standards and Technology, Gaithersburg, MD (2025), https://doi.org/10.6028/NIST.SP.800-226

[30] Nergiz, M.E., Atzori, M., Clifton, C.: Hiding the presence of individuals from shared databases. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. pp. 665–676 (2007)

[31] Nergiz, M.E., Clifton, C.: -presence without complete world knowledge. IEEE Trans. on Knowl. and Data Eng. 22(6), 868–883 (Jun 2010), https://doi.org/10.1109/TKDE.2009.125

[32] NIELSEN, J.: Response times : The three important limits. Usability Engineering (1993), https://cir.nii.ac.jp/crid/1571698600977853312

[33] Olarewaju, T.: 3 reasons why all java developers should consider quarkus (May 2022), https://www.codelikethewind.org/2022/05/03/3-reasons-why-all-java-developers-should-consider-quarkus/

[34] Orlitsky, A., Suresh, A.T.: Competitive distribution estimation: Why is good-turing good. Advances in Neural Information Processing Systems 28 (2015)

[35] Painsky, A.: Convergence guarantees for the good-turing estimator. Journal of Machine Learning Research 23(279), 1–37 (2022), http://jmlr.org/papers/v23/21-1528.html

[36] Pitman, J.: Random discrete distributions invariant under size-biased permutation. Advances in Applied Probability 28(2), 525–539 (1996)

[37] Poddar, R., Wang, S., Lu, J., Popa, R.A.: Practical volume-based attacks on encrypted databases. In: 2020 IEEE European Symposium on Security and Privacy (EuroSP). pp. 354–369 (2020)

[38] Poepsel-Lemaitre, R., Beedkar, K., Markl, V.: Disclosure-compliant query answering. Proc. ACM Manag. Data 2(6) (Dec 2024), https://doi.org/10.1145/3698808

[39] Prasser, F., Bild, R., Eicher, J., Spengler, H., Kohlmayer, F., Kuhn, K.A.: Lightning: Utility-driven anonymization of high-dimensional data. Trans. Data Priv. 9(2), 161–185 (2016)

[40] Prasser, F., Kohlmayer, F., Lautenschläger, R., Kuhn, K.A.: Arx-a comprehensive tool for anonymizing biomedical data. In: AMIA Annual Symposium Proceedings. vol. 2014, p. 984 (2014)

[41] Rajendran, K., Jayabalan, M., Rana, M.E.: A study on k-anonymity, l-diversity, and t-closeness techniques. IJCSNS 17(12), 172 (2017)

[42] Red Hat: Quarkus: Supersonic subatomic java. https://quarkus.io/ (2025), version 3.22.3, Accessed: 2025-11-18

[43] Red Hat: Quarkus, virtual thread support reference. https://quarkus.io/guides/virtual-threads (2025), version 3.22.3, Accessed: 2025-11-18

[44] Shastri, S., Banakar, V., Wasserman, M., Kumar, A., Chidambaram, V.: Understanding and benchmarking the impact of gdpr on database systems. Proceedings of the VLDB Endowment 13(7), 1064–1077 (Mar 2020), http://dx.doi.org/10.14778/3384345.3384354

[45] Skinner, C.J., Elliot, M.J.: A measure of disclosure risk for microdata. Journal of the Royal Statistical Society Series B: Statistical Methodology 64(4), 855–867 (10 2002), https://doi.org/10.1111/1467-9868.00365

[46] Steinke, T., Nasr, M., Jagielski, M.: Privacy auditing with one (1) training run. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 49268–49280. Curran Associates, Inc. (2023), https://proceedings.neurips.cc/paper_files/paper/2023/file/9a6f6e0d6781d1cb8689192408946d73-Paper-Conference.pdf

[47] Supervisor, E.D.P.: Anonymisation - european data protection supervisor https://www.edps.europa.eu/system/files/2021-04/21-04-27_aepd-edps_anonymisation_en_5.pdf

[48] Sweeney, L.: k-anonymity: A model for protecting privacy. International journal of uncertainty, fuzziness and knowledge-based systems 10(05), 557–570 (2002)

[49] Tramer, F., Terzis, A., Steinke, T., Song, S., Jagielski, M., Carlini, N.: Debugging differential privacy: A case study for privacy auditing (2022), https://arxiv.org/abs/2202.12219

[50] Valenzuela, A.: What is data anonymization? techniques, tools, and best practices (2024), https://www.datacamp.com/blog/what-is-data-anonymization, accessed: 2025-11-07

[51] De Capitani di Vimercati, S., Foresti, S., Livraga, G., Samarati, P.: Protecting privacy in data release. In: International School on Foundations of Security Analysis and Design, pp. 1–34. Springer (2011)

[52] Yoo, S., Shin, M., Lee, D.: An approach to reducing information loss and achieving diversity of sensitive attributes in k-anonymity methods. Interactive Journal of Medical Research 1(2), e14 (Nov 2012)

[53] Zayatz, L.V.: Estimation of the percent of unique population elements on a microdata file using the sample. US Bureau of the Census (1991)

[54] Zeng, J., Chua, Z.L., Chen, Y., Ji, K., Liang, Z., Mao, J.: Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In: NDSS (2021)