UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

PAULO ROBERTO PESSOA AMORA

DIFFERENTIALLY PRIVATE CONSENT STORAGE AND ENFORCEMENT

FORTALEZA

2025

PAULO ROBERTO PESSOA AMORA

DIFFERENTIALLY PRIVATE CONSENT STORAGE AND ENFORCEMENT

Thesis submitted to the Programa de Pós-graduação em Ciência da Computação of the Centro de Ciências of the Universidade Federal do Ceará, as a partial requirement for obtaining the title of Doctor in Ciência da Computação. Concentration Area: Ciência da Computação

Advisor: Prof. Dr. Javam de Castro Machado

FORTALEZA

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A541d Amora, Paulo Roberto Pessoa.
Differentially Private Consent Storage and Enforcement / Paulo Roberto Pessoa Amora. – 2025.
106 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2025.
Orientação: Prof. Dr. Javam de Castro Machado.

1. Consent. 2. Data Structures. 3. Differential Privacy. 4. Databases. I. Título.

CDD 005

PAULO ROBERTO PESSOA AMORA

DIFFERENTIALLY PRIVATE CONSENT STORAGE AND ENFORCEMENT

Thesis submitted to the Programa de Pós-graduação em Ciência da Computação of the Centro de Ciências of the Universidade Federal do Ceará, as a partial requirement for obtaining the title of Doctor in Ciência da Computação. Concentration Area: Ciência da Computação

Approved on: 25/08/2025

EXAMINATION BOARD

Prof. Dr. Javam de Castro Machado (Advisor)
Federal University of Ceará (UFC)

Prof. Dr. Angelo Roncalli Alencar Brayner
Federal University of Ceará (UFC)

Prof. Dr. José Maria da Silva Monteiro Filho
Federal University of Ceará (UFC)

Prof. Dr. Sérgio Lifschitz
Pontifical Catholic University of Rio de Janeiro (PUC-RIO)

Profa. Dra. Carmem Satie Hara
Federal University of Paraná (UFPR)

To my family and everyone that supported me through this period. I dedicate this thesis.

ACKNOWLEDGEMENTS

Reaching this point has been both a challenging and incredibly rewarding journey, and I am deeply grateful to everyone who has walked alongside me along the way.

First and foremost, I want to thank my advisor, Javam Machado. Your guidance, patience, and belief in my work gave me the confidence to keep pushing forward, even during the most uncertain moments. Your mentorship has been invaluable both academically and personally, and I am sincerely grateful for the opportunity to learn under your supervision.

To my committee members, José Maria Monteiro, Angelo Brayner, Sérgio Lifschitz, and Carmen Hara, thank you for your thoughtful questions, encouragement, and the time you've dedicated to evaluate my research through our checkpoints. Your perspectives helped me refine and improve my work in ways I couldn't have achieved alone.

To my family — thank you for being my foundation. To my parents Synthia and Paulo: your love and support have been the constant that kept me grounded, even when I felt overwhelmed. Your sacrifices and encouragement gave me the space and strength to chase this dream. To my girlfriend Greyce, thanks for being there for me all those years, through thick and thin.

To the CoreDB cell, Daniel Praciano and Ítalo Abreu — thank you for the discussions, shared stress, and the laughter that kept me sane. I'll carry those memories with me far beyond this degree.

To my contemporaries, with ”one foot in the grave”, Eduardo Neto (a.k.a. Mega) and Serafim Filho, for the support and insightful talks in between our final year of Ph.D.

To my current and former lab mates, André Luis Mendonça, Victor Farias, Malu Maia, Felipe Timbó, Iago Chaves, Lucas Falcão, Antônio José (AJ), Tiago Nascimento, Israel Vidal, Elvis Teixeira, Fernando Dione, Bruno Leal, Rafael Sombra, Daniel Rezende, Júlio César, Elsa Martins, Ubiratan Soares, Gerbson Lima, Rebeca Benevides. Thanks for all the support in my research and project activities in the lab.

To LSBD, especially the administrative section, which keeps the engine running: Renata Santana, Renata Torres and Profa. Roselia Machado, for providing me with the infrastructure and support to perform my research as well as I could, without any obstacles in terms of tools and location, including funding to present our research abroad.

To IFCE, represented by our deans Virgílio Araripe and Wally Menezes, the directors of Campus Cedro, my first placement: Fernando Melo and Gleydson Bastos, and the directors of

Campus Mombaça, my current placement: Eudes Bandeira and Canuto Diógenes for the support provided from schedule adjustment for my classes, to financial support to attend conferences.

To my friends and colleagues at IFCE: Jamires Costa, Marisergio Alves, Saulo Oliveira, Suzana Mafra, José Carlisson, Demócrito Sobreira, Cinthya Morais, Henrique Andrade, Felipe Arrais, Rubens Cainan, Victor Lemos, Felipe Vasconcelos, Aline Oliveira, Paulo Vitor, Daniel Fonseca, Madalena Queiroz, Mário Lima, Tiago Freire and everyone else. Thank you for making the weekly trips worthwhile and directly and indirectly supporting my research.

Finally, to everyone who, in big or small ways, helped me get here: thank you. This thesis is not just mine — it's a reflection of the many people who believed in me when I needed it most.

“O sonho é que leva a gente para frente. Se a gente for seguir a razão, fica aquietado, acomodado.”

(Ariano Suassuna)

ABSTRACT

The increasing adoption of data protection regulations, such as GDPR, CCPA, and LGPD, has made the storage and enforcement of user consent a critical requirement for database systems. Consent metadata, however, introduces new challenges: it must not only enable precise enforcement of user choices but also remain private, since the opt-in/opt-out status itself can leak sensitive information. This thesis addresses this dual challenge by proposing novel data structures that both represent and protect consent metadata under rigorous privacy guarantees. First, we introduce Purpose Filter, a filter-based data structure that encodes purpose metadata with a zero false positive rate, ensuring that unauthorized access requests are never granted. While Purpose Filter improves storage efficiency and query integration, its probabilistic nature is insufficient to protect against set-reconstruction attacks. To overcome this, we develop the Differentially Private Counting Bloom Filter (DPCBF), a new construction that perturbs counters with integer-valued geometric noise, providing formal differential privacy guarantees while maintaining practical utility. Building on these foundations, we design the Differentially Private Purpose Filter (DPPF), which combines consent enforcement with differential privacy without violating consent constraints. We conduct extensive experimental evaluations, including analyses of filter behavior, comparisons with existing DP-Bloom Filters, resistance to reconstruction attacks, and applications in machine learning with consented datasets. Results show that our structures effectively balance privacy, utility, and compliance, offering a practical pathway for privacy-preserving consent enforcement in modern database systems.

Keywords: Consent; Data Structures; Differential Privacy; Databases

RESUMO

A adoção crescente de leis e regulações de proteção de dados, como GDPR, CCPA e LGPD, tornou o armazenamento e garantia do consentimento de usuários um requisito crítico para sistemas de banco de dados. Os metadados de consentimento, entretanto, introduzem novos desafios: precisam permitir não apenas a garantia precisa das escolhas assim como permanecer privados, pois a seleção de aceite/recusa por si só já pode vazar informação sensível sobre o indivíduo. Essa tese aborda este duplo desafio propondo novas estruturas de dados que tanto representam como protegem os metadados de consentimento, sob garantias formais de privacidade. Inicialmente, introduzimos o Purpose Filter, uma estrutura de dados baseada em filtro que codifica estes metadados com uma taxa de falso positivo de 0, garantindo que acessos a dados não consentidos nunca sejam permitidos. Enquanto o Purpose Filter melhora a eficiência de armazenamento e a integração de consultas, sua natureza probabilística é insuficiente para proteger contra ataques de reconstrução de conjunto. Para superar esta deficiência, desenvolvemos o Differentially Private Counting Bloom Filter (DPCBF), uma nova construção que perturba contadores inteiros com ruído geométrico, provendo garantias formais de privacidade diferencial enquanto mantém utilidade prática. Sobre estas fundações, projetamos o Differentially Private Purpose Filter (DPPF), que combina a garantia de consentimento com privacidade diferencial, sem violar restrições de consentimento. Conduzimos uma avaliação experimental extensa, incluindo análises de comportamento do filtro, comparações com já existentes DP-Bloom Filters, resistência a ataques de reconstrução, e aplicações em aprendizado de máquina com dados consentidos. Os resultados mostram que nossas estruturas equilibram privacidade, utilidade e adequação às restrições, oferecendo uma forma prática para garantia de consentimento com preservação de privacidade em sistemas de bancos de dados modernos.

Palavras-chave: Consentimento; Estruturas de dados; Privacidade Diferencial; Bancos de Dados

LIST OF FIGURES

Figure 1 – Consent and personal data providing 17
Figure 2 – Consent form, highlighting purposes and user's option to each one 18
Figure 3 – Consent metadata (right) compared to personal data (left) 19
Figure 4 – Data workflows 26
Figure 5 – Bloom Filter 29
Figure 6 – Counting Bloom Filter 30
Figure 7 – Differential Privacy example 31
Figure 8 – Two-sided Geometric Distribution 32
Figure 9 – Geometric Mechanism overlapped with true values 33
Figure 10 – Query workflow in Sieve 38
Figure 11 – Process of extraction of true positives from a Counting Bloom Filter 41
Figure 12 – Bloom filter composed of elements $x1, x2, x3$ that admits 3 false positives: $\upsilon1, \upsilon2, \upsilon3$. 45
Figure 13 – Bloom filter before and after BLIP algorithm. 46
Figure 14 – Distribution W for $k = 3$, $m = 2^{19}$ and $|P| = 10000$ 47
Figure 15 – Purpose Filter construction. 52
Figure 16 – Probe example 54
Figure 17 – DP-Purpose Filter construction (reads left-to-right) 73
Figure 18 – DP-Purpose Filter probe 74
Figure 19 – Experiments for fixed size 79
Figure 20 – Experiments for fixed FPR_0 80
Figure 21 – Comparison with other works 82
Figure 22 – DPCBF shape for different values of ε 84
Figure 23 – $k = 2$ 86
Figure 24 – $k = 3$ 86
Figure 25 – $k = 4$ 86
Figure 26 – $k = 5$ 86
Figure 27 – $k = 6$ 87
Figure 28 – $m = 2^{18}$ 87
Figure 29 – $m = 2^{19}$ 88
Figure 30 – $m = 2^{20}$ 88

Figure 31 – $m = 2^{21}$ 88
Figure 32 – Inserted size $|P| = 10$ 89
Figure 33 – Inserted size $|P| = 100$ 89
Figure 34 – Inserted size $|P| = 1000$ 89
Figure 35 – Inserted size $|P| = 10000$ 90
Figure 36 – Inserted size $|P| = 100000$ 90
Figure 37 – Inserted size $|P| = 10000$ 94
Figure 38 – Inserted size $|P| = 30000$ 95
Figure 39 – Inserted size $|P| = 55000$ 95
Figure 40 – Inserted size $|P| = 70000$ 95
Figure 41 – Inserted size $|P| = 90000$ 95

LIST OF TABLES

Table 1 – Summary of the existing works. 48
Table 2 – Number of layers and pair sizes for different fixed sizes. 81
Table 3 – Number of layers and pair sizes for different fixed FRRs. 81
Table 4 – Jaccard similarity for different ε values and constant $m = 65536$. 91
Table 5 – Jaccard similarity for different ε values and constant $\frac{|P|}{m}$. 91
Table 6 – Jaccard similarity for different ε values and constant $m = 65536$. 92
Table 7 – MAE for the polling scenario with different ε values. 93
Table 8 – Exploratory analysis of the datasets. 98
Table 9 – Results for mean imputation 99
Table 10 – Results for median imputation 100
Table 11 – Results for mode imputation 100
Table 12 – Results for mean imputation/SMOTE 101
Table 13 – Results for median imputation/SMOTE 101
Table 14 – Results for mode imputation/SMOTE 101

LISTA DE ALGORITMOS

Algorithm 1 – Purpose Filter Constructor function 50
Algorithm 2 – Purpose Filter probing function 53
Algorithm 3 – DP Counting Bloom Filter constructor function 61
Algorithm 4 – DP Counting Bloom Filter query function 62
Algorithm 5 – DPPF Constructor function 72

CONTENTS

1 INTRODUCTION . . . 17
1.1 Problem Statement . . . 22
1.2 Hypothesis . . . 22
1.3 Contributions . . . 23
1.4 Thesis Organization . . . 24
2 THEORETICAL BACKGROUND . . . 25
2.1 Purpose and Consent Metadata . . . 25
2.2 Filters . . . 28
2.2.1 Bloom Filter. . . . 28
2.2.2 Counting Bloom Filter. . . . 29
2.3 Differential Privacy . . . 30
2.3.1 Differential Privacy Mechanisms . . . 31
2.3.2 Differential Privacy Properties . . . 33
2.3.3 Summary . . . 34
3 RELATED WORK . . . 35
3.1 Approaches for storing and representing consent data . . . 35
3.1.1 Hippocratic Databases (AGRAWAL et al., 2002) . . . 36
3.1.2 Sieve (PAPPACHAN et al., 2020) . . . 37
3.1.3 GDPRBench (SHASTRI et al., 2020) . . . 38
3.2 Attacks on filters . . . 40
3.3 Privacy on set membership data . . . 42
3.3.1 Private Set Intersection (FREEDMAN et al., 2004) . . . 42
3.3.2 Differentially Private Set Intersection and Union (XUE et al., 2017) . . . 43
3.4 Approaches for private filters . . . 43
3.4.1 Bianchi et. al. (BIANCHI et al., 2012) . . . 44
3.4.2 BLIP (ALAGGAN et al., 2012) . . . 44
3.4.3 DPBloomFilter (KE et al., 2025) . . . 45
3.4.4 Discussion . . . 47
3.5 Summary . . . 47
4 PURPOSE FILTERS . . . 49
4.1 Filter Construction . . . 49

4.2 **Filter Probing** . . . 53
4.3 **Filter Maintenance** . . . 55
4.4 **Information Loss** . . . 55
4.5 **Space allocation** . . . 56
4.6 **Costs** . . . 57
4.7 **Optimizing filter construction** . . . 57
4.8 **Purpose and Consent Modeling using Purpose Filter** . . . 58
4.9 **Summary** . . . 59
5 **DIFFERENTIALLY PRIVATE COUNTING BLOOM FILTERS** . . . 61
5.1 **Privacy Guarantees** . . . 62
5.2 **Utility Evaluation** . . . 66
5.3 **Summary** . . . 68
6 **DIFFERENTIALLY PRIVATE PURPOSE FILTER** . . . 70
6.1 **Filter Construction** . . . 70
6.2 **Filter Probing** . . . 73
6.3 **Information Loss** . . . 75
6.4 **Privacy Guarantees** . . . 75
7 **EXPERIMENTAL EVALUATION** . . . 77
7.1 **Purpose Filter** . . . 77
7.1.1 ***Evaluation Setup*** . . . 77
7.1.2 ***Fixed size*** . . . 78
7.1.3 ***Fixed*** FPR_0 . . . 80
7.1.4 ***Experiments for number of layers and layer sizes*** . . . 81
7.1.5 ***Comparison with other works*** . . . 82
7.2 **Differentially Private Counting Bloom Filters** . . . 83
7.2.1 ***DPCBF Shape*** . . . 84
7.2.2 ***Comparison with DPBloomFilter*** . . . 84
7.2.2.1 *Varying* k . . . *85*
7.2.2.2 *Varying* m . . . *87*
7.2.2.3 *Varying* $|P|$ . . . *88*
7.2.2.4 *Discussion* . . . *89*
7.2.3 ***Attack resistance*** . . . 90

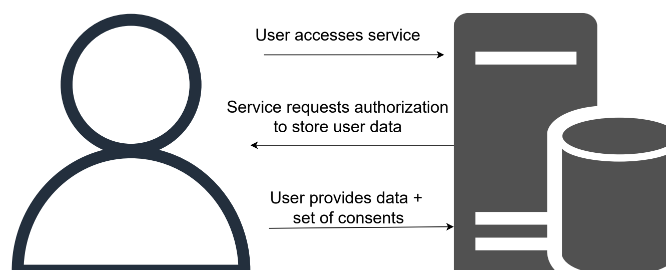
7.2.4 *Application: Polling* 92
7.3 **Differentially Private Purpose Filter** 94
7.3.1 *Comparison of layer types* 94
7.4 **Application: ML training** 96
7.4.1 *Application and dataset* 96
7.4.2 *Setup and workflow* 97
7.4.3 *Exploratory analysis* 98
7.4.4 *Results* 99
8 **CONCLUSION** 102
8.1 **Summary of results** 102
8.2 **Future Works** 103
REFERENCES 105

1 INTRODUCTION

Personal data has become a first-class citizen in the domain of user applications. As a result, government agencies have acted to balance the power dynamic between users and companies, empowering users by providing rights over any ceded data through regulations such as GDPR (General Data Protection Regulation, 2016), CCPA (CCPA, 2018), and LGPD (LGPD, 2018). This data can be used to extract information and insights in analytical scenarios. Online Analytical Processing (OLAP) is characterized by the large volume and immutability of data.

These regulations require that the user who provides the data must provide consent, in a well-informed manner, for what purpose applications may process their data. Database Management Systems (DBMS) usually store data collected through online applications. The Database Administrator (DBA) is responsible for modeling how data will be stored and what should be stored. They are responsible for the correct modeling of stated purposes and user consent. Alongside personal data, consent metadata must be stored to enable enforcement, i.e., we must know if the user authorizes or denies access to their data. Figure 1 and Figure 2 show how this relationship happens. When a user gives their data, they also provide a set of consents, each consent can be modeled as a double (purpose, value), as seen on figure 2. Let us take the 5 purposes shown in Figure 2 as an example, where the user can choose to opt-in or opt-out for each purpose. The result for the shown options is that you would have {user1: {purpose1, no}, {purpose2, no}, {purpose3, no}, {purpose4, no}, {purpose5, no}}, meaning that the user does not consent to share their data for any of the purposes. If any button was set to yes, the change would be reflected in the set. As mentioned before, this consent option has to be respected and enforced; therefore, no personal data from user1 will be returned if a querier requests data for any of these purposes.

Figure 1 – Consent and personal data providing



Source: elaborated by the author.

Figure 2 – Consent form, highlighting purposes and user's option to each one

Purposes | Features | Vendors

Below, you will find a list of the purposes and special features for which your data is being processed. You may exercise your rights for specific purposes, based on consent or legitimate interest, using the toggles below.

Consent | Legitimate interest

Purposes

- Store and/or access information on a device — Off
- Use limited data to select advertising — Off
- Create profiles for personalised advertising — Off
- Use profiles to select personalised advertising — Off
- Measure advertising performance — Off
- Understand audiences through statistics or combinations of data from

Accept all | Reject all | Save

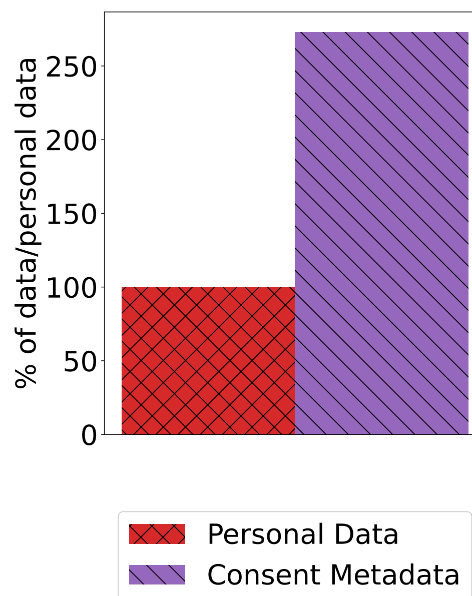
Source: nytimes.com (accessed on 06/10/25)

Consents can be enforced before, during, or after query processing. A common approach to facilitate retrofitting and avoid direct intervention into the DBMS is to use middlewares, as we see in many works (PAPPACHAN *et al.*, 2020; KONSTANTINIDIS *et al.*, 2021; RIZVI *et al.*, 2004). Through middleware, consent enforcement may occur before query processing, via query rewriting in the middleware, or after, as the middleware is capable of filtering non-compliant results before returning them to the querier/application. One gap, as identified in (PRACIANO *et al.*, 2022), is that if someone gains direct access to the underlying database, there are no safeguards related to consent-abiding query answering, since this logic is stored and applied by the middleware. To enforce consents during query processing, one can use views and stored procedures, allied with access control mechanisms like role-based access control, or design specific query operators that use consent information during their execution.

Storing each user's consent data presents a problem where the amount of metadata can overcome the actual data, a problem highlighted in (SHASTRI *et al.*, 2020), called metadata explosion, in which added metadata scaled up to almost 6 times the amount of personal data. This definition encompasses metadata related to all scopes of the legislation, such as the right to access, the right to be forgotten, among others.

We observe that consent metadata, depending on how it is represented and stored, may be a large contributor to this problem. Figure 3 is an excerpt from the experiments, showing that a given approach may increase consent metadata to be almost 300% the storage amount of data, presenting an overhead on storage space. Therefore, we must be mindful of how to represent this data, also avoiding overheads on consent enforcement, which may happen on query processing.

Figure 3 – Consent metadata (right) compared to personal data (left)



Source: elaborated by the author.

Purpose-based access control is not a new problem, with works like (BYUN; LI, 2008; AGRAWAL *et al.*, 2005; RIZVI *et al.*, 2004) providing solutions to this problem. However, they store this metadata directly in the database, using tables. More recently, with the legal support provided by new regulations, new works such as (SHASTRI *et al.*, 2020; PAPPACHAN *et al.*, 2020; DESHPANDE, 2021; KRASKA *et al.*, 2019) discuss other forms of storing this metadata, either as surrogate keys or alongside data, internal to the database.

To exemplify, take a scenario where a user does a health checkup in a hospital and needs to share medical data with two businesses, a pharmacy and a health insurance company. This medical data is composed of tests, their results, and their prices. The hospital requires access to the tests' results to plan medicine purchases. The insurance company needs to know how much the tests cost to cover, but they shouldn't know about the results. The user states that he consents to share the tests' results with the pharmacy, but only the tests' costs with the company. The hospital is responsible for enforcing the user's consent, sharing only appropriate data with each party.

These consent data may be considered sensitive. Consider a scenario where an adversary may know that a specific individual provided their data to some database, not consenting to this data to be used by third parties, for example. Upon obtaining access, the adversary poses queries where this data would be included, but since consent was not given, it is not returned to the adversary. Therefore, the adversary may infer that consent is not given, and knowledge of the consent option of the individual could lead to harm. Therefore, it may not be sufficient to only store this consent data, but to offer privacy guarantees to this data.

We argue that we can leverage filters as a consent data representation. Filters are a probabilistic data structure that maps a set to a fixed-size array through hash functions. From our healthcare example, we could create a filter for each purpose (diagnosis, coverage of costs), and add consent data for each user in the filter. Whenever a query arrives, the purpose is identified, and the filter is accessed to determine from which registers' data should be retrieved.

The probabilistic nature of filters can make the query answers approximate. In an OLAP scenario, this may happen, and we can exploit this property to provide some sort of privacy protection regarding the consents and whoever provided them in the data structure. Without any modifications, filters occur in **false positives**. This fact alone could be a deterrent to using filters as our consent data representation, because false positives mean that a user's opt-out may be misinterpreted as an opt-in, violating their consent. However, we will show that we can achieve a filter with 0 **false positive rate** (FPR), although, as is the nature of filters, we still have uncertainty in the results, but now on the side of **false negatives**. A **false negative rate** (FNR) is acceptable from the perspective of consent enforcement, because it means that someone who consented that their information is shareable, won't be shared. This is not considered a consent violation because no rights are being breached.

So far, we have achieved a consent representation that supposedly protects users and

their consents. However, the filter's embedded uncertainty is shown to be insufficient to provide this protection. Works like (REVIRIEGO *et al.*, 2023a; REVIRIEGO *et al.*, 2024; REVIRIEGO *et al.*, 2023b) show that it is possible to reconstruct the original set used to construct a filter. This is called a **set reconstruction attack**. For each user in a universe, the attacker may be able to identify whether each user consented or not to their data being used. This means we are not yet in the clear regarding privacy. From our example, if the adversary knows that a specific individual provided their data to some database, they can pose queries to the filter using knowledge of other users and attempt to reconstruct the original set, identifying whether the individual consented or not to their data being used. If the adversary succeeds, they can infer that the individual did not consent to share their data with the insurance plan, and based on that negative, discriminate against this user in terms of coverage.

From the previous example, we can see that before the data itself, knowledge of a user's consent option is sensitive and should be protected. Taking cues from social context, where denial or refusal to provide information can be considered an admission of guilt or fault, we can see that the same applies to consent data. If an adversary knows that a user did not consent to share their data, they may infer that the user has something to hide, and this knowledge can be used against them. Therefore, protecting this choice leads to overall privacy of users'.

One way to protect this data would be by using cryptography. However, while cryptography could provide perfect protection, it would have an added cost on processing, but more importantly, it would render the data useless. Having to decrypt this consent metadata to enforce it may also introduce a weak link, where the real metadata is unprotected during a step of the enforcement process. Nevertheless, we may also query this metadata for statistics, something that cryptography also prevents. Intending to still have utility for this data, we move towards Differential Privacy.

Differential privacy (DP) (DWORK, 2006) has emerged as a robust privacy definition, becoming the standard for data release under strong privacy guarantees. The main idea behind DP is that an analysis is determined by a randomized algorithm, also known as a mechanism, that computes private information and returns a randomized answer sampled from a probability distribution. In the literature, the primary DP setups are the global DP (DWORK, 2006) and the local DP (LDP) (DUCHI *et al.*, 2013). The key difference between them lies in the nature of the data curator. In the global setting, it is assumed that there exists a trusted data curator that has indiscriminate access to the complete data and is responsible for releasing it after a differentially

private procedure. In other words, the data curator is the responsible entity for ensuring privacy. In the local setting, the data curator is untrustworthy. Therefore, each user is responsible for protecting their data by applying a privacy algorithm before sending it to the curator.

In this work, we assume that the DBMS, which stores raw data and is a central point of processing, is trustworthy, since it also stores the rules and models of the consent and purpose data; therefore, using the global DP model.

1.1 Problem Statement

Given a set of unique identifiers for each user (e.g. user id) for a same purpose, split into two sets based on their consent option, namely *opt-in* and *opt-out*, and considering that consents' value may reveal unintended information about the owners, i.e., reidentification of a user in a set may provide extra knowledge about this user, our research question is defined as "*How to represent this consent information and store it, protecting the users?*"

Problem Definition: *Consider a set of identifiers U for a given purpose, split in opt-ins (P) and opt-outs (N). Our goal is to release a data structure that allows for publishing and set membership queries while ensuring protection to the identifiers inserted in this data structure and providing usefulness.*

In this problem, we have two main characteristics: (I) the set representation should not incur a heavy space overhead compared to plain storage of purposes in tuples/tables.(II) The set representation must be protectable, meaning that some uncertainty is desirable. The data structure must be resistant to set reconstruction attacks, making it hard enough to certainly allow for identifying individuals that are part of either set. For (I), many useful metrics, such as storage space, incorrect results, and growth behavior, should be similar to naive approaches. For (II), we are interested in the impact of each parameter in comparison to an unperturbed filter. Measuring the error included by decreasing the added noise to the filter and verifying the impact of changing the filter properties, such as filter size, number of inserted items and the quantity of mappings for each item, we can determine and suggest configurations for different setups.

1.2 Hypothesis

Given a universe of identifiers U, partitioned into opt-in (P) and opt-out (N) subsets such that $U = P \cup N$, one can construct a data structure that (i) enables useful representation

and enforcement of consent, (ii) ensures privacy for members of both *P* and *N*, and (iii) achieves these guarantees while incurring only low storage overhead relative to the size of *U*. That is, an attacker cannot identify the *P* and *N* sets with high probability by attacking the data structure while maintaining usefulness for the consent data.

1.3 Contributions

To address the mentioned concerns, we initially propose Purpose Filter, a data structure composed of layers of filters that does not allow false positives. As our investigation revealed, the uncertainty of filters is insufficient to protect the elements of the *P* and *N* sets. For filters, differential privacy was only applied in Bloom Filters, so far. Since our structure can use one of several filter types, we expand our options by exploring differential privacy for Counting Bloom Filters. Finally, we show that directly applying a DP mechanism after construction may introduce wrong behavior in the data structure; therefore, we design an appropriate form to add DP to Purpose Filter, which still applies for any type of filters that can be protected by differential privacy mechanisms. In summary, the main contributions of this work are as follows:

1. We first introduce Purpose Filter, a 0 false positive rate filter data structure that leverages the nature of purpose data to encode it with some uncertainty, by layering filters and narrowing the false negative rate.
2. We investigate differential privacy techniques for filters, based on exposed vulnerabilities, proposing a new differentially private filter, called Differentially Private Counting Bloom Filter, to be used as layers of the Purpose Filter.
3. We apply differential privacy to Purpose Filter, creating a Differentially Private Purpose Filter.
4. We perform experimental analysis on all contributions, asserting effectiveness both in representing consents and in utility as noise is introduced.

The main contributions described in this thesis are also presented in the following published papers:

- P. R. P. Amora, F. D. B. S. Praciano , J. C. Machado. **Purpose Filter: A space-efficient purpose metadata storage.** *XXXIX Conference on Data and Applications Security and Privacy. 2025.*
- Praciano, F. D. B. S.; Amora, P. R. P.; Abreu, I. C.; Machado, J. C. **Purpose Scan: A purpose-aware access method.** In: *Heterogeneous Data Management, Polystores, and*

Analytics for Healthcare - VLDB 2022 Workshops, Poly and DMAH. 2022.

- MACHADO, J. C.; AMORA, P. R. P. **The impact of privacy regulations on DB systems.** *Journal of Information and Data Management, v. 12, n. 5, 2021.*

A tutorial was also published and presented to disseminate and broaden the scope and impact of our research:

- Machado, J. C.; Amora, P. R. P.; Praciano, F. D. B. S. **Purpose and Consent Enforcement in DBMS.** *Anais Estendidos do XXXIX Simpósio Brasileiro de Bancos de Dados*

The contributions also resulted in the following work to be submitted:

- Amora P. R. P., Chaves I. C., Machado J. C. **Protecting Consent Metadata Using Differential Privacy** *(VLDB 2026).*

1.4 Thesis Organization

The rest of this document is described as follows. Chapter 2 presents an overview of several concepts to help understand this work, such as filters, with more detail on the types of filters used in this work and the properties and settings of DP. Chapter 3 presents some studies on consent and purpose data storage and representation, how filters may be attacked and the inserted set may be reconstructed, and finally, techniques to protect this data under DP guarantees. Chapter 4 details the Purpose Filter data structure. We then present an algorithm to protect Counting Bloom Filters in Chapter 5. In Chapter 6, we discuss how directly applying DP to Purpose Filter can violate consent restraints, and propose a new construction algorithm that guarantees both differential privacy and correct consent enforcement. Chapter 7 shows our experimental evaluation over three sections; of Purpose Filter by itself, changing layer configurations and varying over the proportion of opt-ins vs. opt-outs; of DPCBF, showing its behavior in comparison to a non-perturbed filter, also comparing it to a DPBloomFilter, visualizing the data structure and added noise, replicating the set reconstruction attack, to verify the added protection, and show a polling scenario in which this structure can be used; of DPPF, showing the impact on false negative rate based on the amount of noise inserted. We also compare the FNR for DP-Bloom Filter layers and DP-Counting Bloom Filter layers, wrapping up with a machine learning application in which we simulate consented datasets, to estimate the impact. Finally, Chapter 8 concludes this thesis and discusses future work directions.

2 THEORETICAL BACKGROUND

In this chapter, we describe the theoretical background of this thesis, which is divided into three main parts: purpose and consent data access and representation, filters as a data structure, and differential privacy. The first part describes the purpose and consent metadata, which is the main focus of this thesis. The second part describes filters, which are the data structure used to store the purpose and consent metadata. The third part describes differential privacy, which is a privacy-preserving mechanism that can be used to protect the purpose and consent metadata.

2.1 Purpose and Consent Metadata

To model our filter, we first state our understanding of what is purpose and what data is necessary to enforce it. We use the following entities, definitions, and claims.

From GDPR (General Data Protection Regulation, 2016), we define three entities:

Data Owner: The party who provides the data. They have all rights pertaining to data processing and usage and may object to this use. Therefore, they can consent to which purposes the Controller assigns to their data.

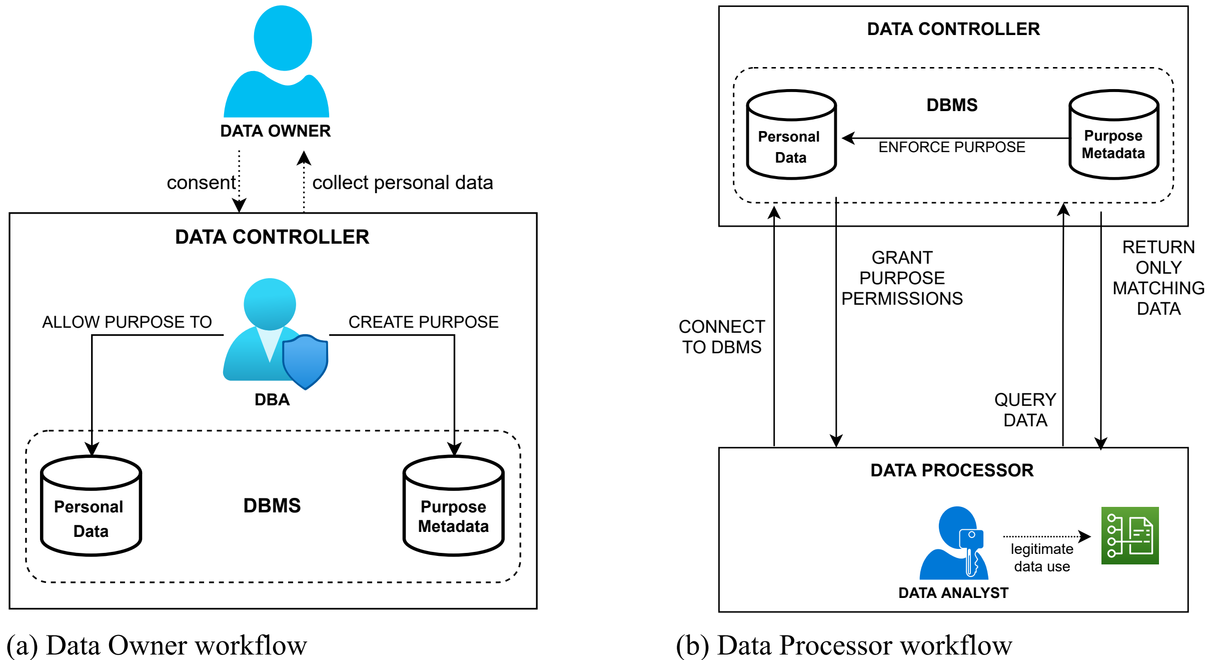
Data Processor: The party that intends to process data from the Data Owners. They must clearly state what ends they will use the retrieved data for and how they will consume it.

Data Controller: The party that stores the data collected from Data Owners and provides them to Data Processors. According to the Owners' requests, they manage the access permissions of personal data, actively allowing or denying access to data.

Figure 4a shows the relationship between Data Owner and Data Controller. The Data Controller configures the existing purposes, adding the purpose metadata. Whenever the Data Owner provides data to the Data Controller (e.g., sign up for a service, allow the Data Controller to read stored cookies), the Data Controller will present the existing purposes, and the Data Owner will either accept or reject their data usage by Data Processors. The purpose creation process requires active intervention by a DBA. However, the Data Controller can perform the assignment automatically, be it the DBA or a configured DBMS. This is where our filter will be positioned, for storing this purpose and consent data.

Figure 4b shows the relationship between Data Controller and Data Processor. When the Data Processor queries the Data Controller's DBMS, the DBMS links a preconfigured purpose

Figure 4 – Data workflows



(a) Data Owner workflow

(b) Data Processor workflow

Source: elaborated by the author.

to the Data Processor. The DBMS processes queries sent by the Processor and uses the purpose metadata to enforce purpose-based access control over any personal data queried by the Data Processor. Therefore, only data from Data Owners who consented to the processing purpose will be returned to the Data Processor. From the perspective of the data analyst querying the database, this enforcement happens transparently. There are no changes in connecting to the database or querying data.

To provide a more concrete example, John (Data Owner) and Doe (Data Owner) intend to use a social network $\mathcal{S}$. When signing up, they provide their personal data, and $\mathcal{S}$ requests consent for different ends, such as personalized ads, machine learning, and user profiling (Assigned Purposes). John accepts all uses; Doe only accepts machine learning. $\mathcal{S}$ manages all personal data provided to them. $\mathcal{S}$ partners with AnalyticsBC (Data Processor) to help increase its user engagement. AnalyticsBC needs to process personal data from $\mathcal{S}$'s users. AnalyticsBC requests data from $\mathcal{S}$, stating their Processing Purpose as user profiling. $\mathcal{S}$ will only provide personal data from users who consented to have their data used to this end, which means that AnalyticsBC will only see John's data.

From these relationships, we can establish some definitions used throughout this work.

Definition 1: A purpose is a finality in which data will be processed, e.g., Sales,

Advertisement, Business Intelligence.

Definition 2: Assigned Purpose (AP) is the stated purpose that the Data Owner allows their data usage. APs are managed by the Data Controller and stored alongside the data through a mechanism explained further. APs are assigned to a data instance (relation, tuple, or tuple attribute), and a data instance may have more than one AP.

Definition 3: Processing Purposes (PP) are the declared ends for which the Data Processor intends to use personal data. The Data Processor declares these upon requesting data from the Data Controller.

From these definitions, we can establish the following sets known by the Data Controller:

- OS = The set of APs in personal data provided by the Owner.
- PS = The set of PPs present in the Processor's request.

To ensure that the Owner's consents are being respected, the Data Controller defines and uses a purpose verification function $PV(OS, PS)$ to verify that $PS \subseteq OS$. The Data Controller must return only data in which the processing purposes (PS) match the assigned purposes (OS). Therefore, to consider a DBMS implementation of that purpose verification function, it must comply with the following claims:

Claim 1. Personal data, which is data provided by a Data Owner, is stored as tuples in one or more relations and has one or more associated AP.

Claim 2. A Data Owner may consent to each purpose individually regarding their personal data. These consents form the OS set.

Claim 3. A Data Processor must state their PP to access personal data. These form its PS set, and this type of access is named purpose-aware access.

Claim 4. If access is attempted to Personal Data and the PP does not match the data's AP, access is denied. In other words, if access is not allowed, it is prohibited.

Consent can also be given in a more granular way, such as a user accepting a purpose for some of their attributes, instead of their whole data.

To ensure that the Owner's consents are being respected, the Data Controller defines and uses a purpose verification function to verify that only allowed tuples return. The Data Controller must return only data in which the processing purposes match the assigned purposes.

From these definitions, we can observe that purpose verification can be defined as a **set membership problem**, where the set is the OS set, and the element to be verified is the PS set.

2.2 Filters

Filters are probabilistic data structures designed primarily for checking if an element is a member of a set. Usually, filters encode variable-sized data in a fixed-size data structure through hash functions. Because of that, collisions may occur, and the filter may yield a false result. Nevertheless, filters are used in many applications, such as caching and encoding values to be communicated.

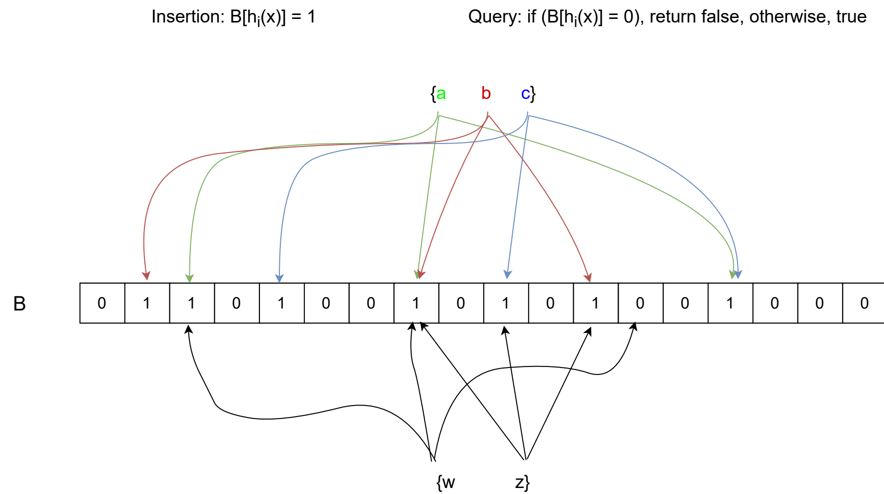
2.2.1 Bloom Filter.

Bloom Filters (BLOOM, 1970) are a probabilistic data structure, composed of a bit array of size m, initialized as zeroes and a set of k hash functions H, where each function $h_1(x), h_2(x), ..., h_k(x)$ map an element x to a position in this bit array by changing the value of the positions $h_1(x), h_2(x), ..., h_k(x)$ to one.

To check if an element is present, one must hash it through all functions in H and verify if all the bits are set to one. If not, we are certain the element was not inserted into the filter. If so, the element may have been added to the filter, since the insertion of other elements could have also changed the value of its positions. This design allows for a data structure that provides no false negatives, but it may yield false positives.

Figure 5 shows a Bloom Filter with 18 bits and 3 hash functions. Elements $\{a, b, c\}$ are added to the filter. Element a is hashed through the three functions, which set the bits at positions 2, 7, and 14 to one. In the same fashion, b is hashed and set bits at positions 1, 7, and 11 to one, and c hashes to 4, 9, and 14. To check if an element is present, it is hashed through the same functions. If all bits are set to one, the element may be present in the filter. For example, we provide two elements $\{w, z\}$ to the filter. Element w hashes to positions 2, 7, and 12. Positions 2 and 7 are set to one, but 12 is set to 0, which means that an element never before was hashed to position 12; therefore, we can be certain that w was never inserted in the filter. Element z hashes to positions 7, 9, and 11, which are all set to one, so the filter returns that z may be present in the filter. Even though z was never inserted, the filter returns that it is present, since other elements

Figure 5 – Bloom Filter



Source: elaborated by the author.

have set the bits at positions 7, 9, and 11 to one. This is an example of a false positive.

Bloom Filters are space-efficient, and the probability of false positives can be controlled by adjusting the size of the bit array and the number of hash functions. The probability of false positives is given by:

$$P_{fp} = \left(1 - e^{-\frac{kn}{m}}\right)^k, \tag{2.1}$$

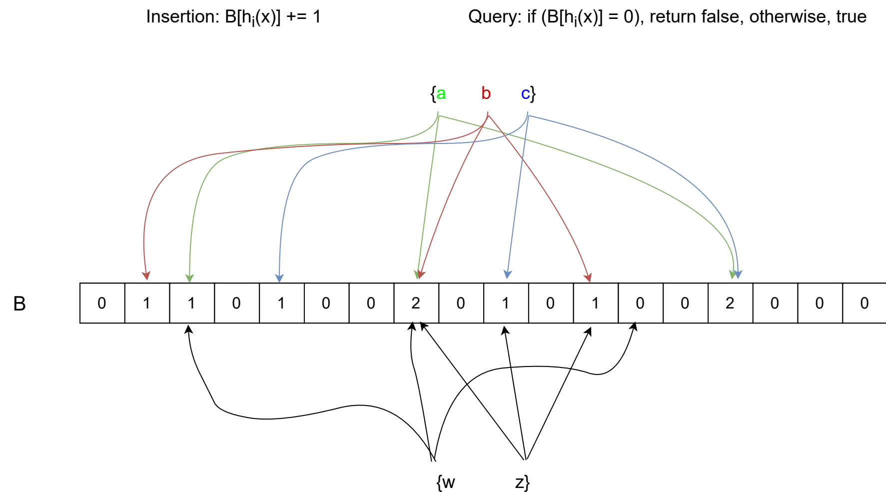
Where k is the number of hash functions, m is the size of the array, and n is the quantity of inserted items in the filter.

The property of false positives is of our interest, since it generates uncertainty whether an element is actually present in the filter.

2.2.2 Counting Bloom Filter.

One limitation of Bloom Filters is that since they only store bits, it is not possible to estimate how many elements are present in the filter. Counting Bloom Filters (CBF) (BONOMI *et al.*, 2006) store counters instead of bits, allowing for a fast approximate membership check, like Bloom Filters, with added functionality. Other operations occur similarly, since they also have a set of hash functions that verify if an element is present. Counting Bloom filters also allow for accurate estimation of the inserted set size, because each hash function modifies exactly one position.

Figure 6 – Counting Bloom Filter



Source: elaborated by the author.

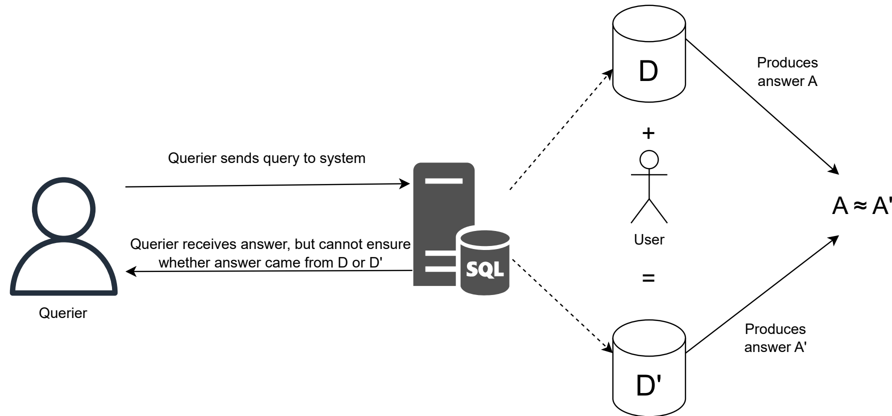
Figure 6 repeats the example from Figure 5, but now using a Counting Bloom Filter. The filter is composed of a counter array of size 18, initialized to zero. The elements $\{a, b, c\}$ are added to the filter, which increments the counters at positions 2, 7, and 14 for a, 1, 7, and 11 for b, and 4, 9, and 14 for c. To check if an element is present, it is hashed through the same functions. If all counters are greater than zero, the element may be present in the filter. For example, we provide two elements $\{w, z\}$ to the filter. Element w hashes to positions 2, 7, and 12. Counters at positions 2 and 7 are greater than zero, but position 12 is zero, which means that never before an element was hashed to position 12, therefore, we can be certain that w was never inserted in the filter. Element z hashes to positions 7, 9, and 11, which are all greater than zero, so the filter returns that z may be present in the filter. Even though z was never inserted, the filter returns that it is present, since other elements have set the counters at positions 7, 9, and 11 to greater than zero. This is an example of a false positive.

CBFs also allow for the estimation of the number of elements present in the filter, which is given by the sum of all counters divided by the number of hash functions. If only unique elements are inserted, the estimated size is equal to the number of elements inserted.

2.3 Differential Privacy

Differential privacy (DP) (DWORK, 2006) is a robust privacy definition that has become the standard for data release under strong privacy guarantees. In summary, it states that any answer to a query occurs with similar probability regardless of the presence or absence of

Figure 7 – Differential Privacy example



Source: elaborated by the author.

any individual in the dataset, as defined as follows:

Definition 2.1 (ε-Differential Privacy (DWORK, 2006))**.** A randomized algorithm $\mathcal{A}$ satisfies ϵ-differential privacy if for any two neighboring datasets D, D' and for any output $O \subseteq Range$ $(\mathcal{A}), Pr[\mathcal{A}(D) = O] \leq e^{\varepsilon} Pr[\mathcal{A}(D') = O]$.

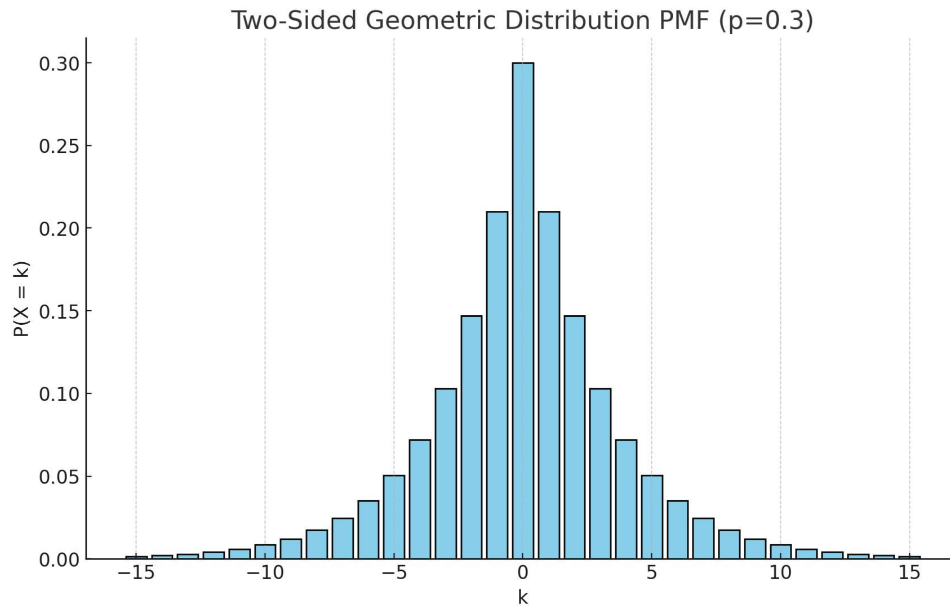
Note that D and D' are neighbors if they differ in at most one single record. A smaller ε corresponds to a stronger privacy-preserving guarantee, while a higher ε provides better data utility.

Figure 7 illustrates the concept of differential privacy. The figure shows a querier interacting with a system, which is connected to two datasets, D and D'. A querier submits a query to the system, which returns an answer. The querier should not know whether the answer was generated from D or D'. The system ensures that the probability of receiving a specific answer is similar for both datasets, regardless of the presence or absence of any individual record. This is achieved by adding noise to the query results, which is controlled by the privacy parameter ε. Therefore, the answers generated from the two datasets are similar, providing privacy guarantees.

2.3.1 Differential Privacy Mechanisms

The randomized algorithm $\mathcal{A}$ is also referred to as a mechanism, which is a way of achieving DP. For numerical queries, DP can be achieved by various mechanisms, such as Laplace (DWORK, 2006) and geometric (GHOSH *et al.*, 2009) mechanisms. The Laplace mechanism is recommended for queries that output real values, while the geometric mechanism is recommended for queries that output integer values. Our approach follows the geometric

Figure 8 – Two-sided Geometric Distribution



Source: elaborated by the author.

mechanism to add integer noise to the values stored by the CBF.

Definition 2.2 (Global Sensitivity (DWORK *et al.*, 2006))**.** The global sensitivity of a query Q is the maximum l_1 distance between the outputs of Q on any two neighboring datasets D e D', given by $\Delta Q = \max_{D,D'} ||Q(D) - Q(D')||_1$.

The geometric mechanism adds integer noise to the true query answers following the two-sided geometric distribution, defined as:

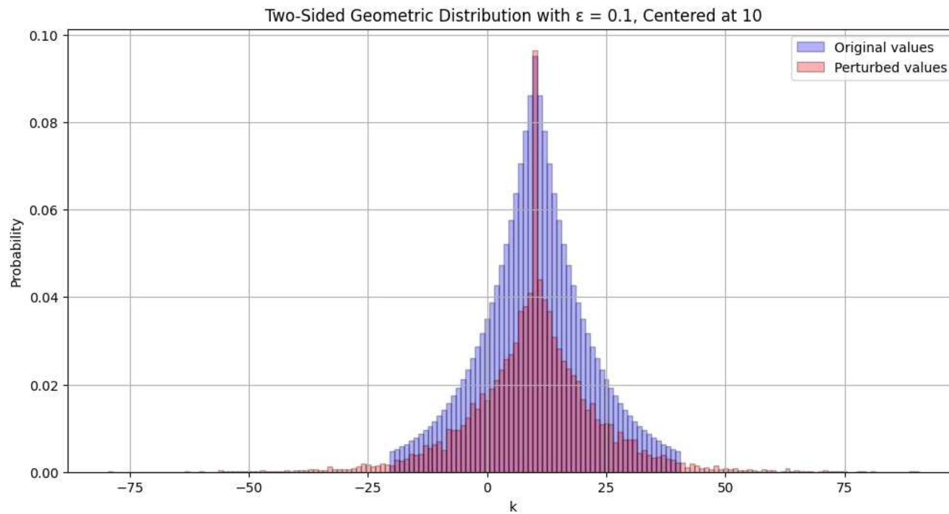
Definition 2.3 (Two-sided Geometric Distribution)**.** A random variable X distributed as a two-sided geometric distribution, with mean 0 and $\alpha \in [0, 1]$, has a probability mass function $P(X = x) = \frac{1-\alpha}{1+\alpha}\alpha^{|x|}$.

Figure 8 illustrates the two-sided geometric distribution with $\alpha = 0.3$, meaning that the probability of sampling a 0 from this distribution is 30%, a probability of 20% to sample either ± 1, and so on. The distribution is symmetrical in the center, and as expected, since it is a discrete version of the Laplace distribution, the values' probabilities decrease exponentially as we move away from the center.

We denote $Geom(\frac{\varepsilon}{\Delta Q})$ the two-sided geometric distribution with mean 0 and $\alpha = e^{-\frac{\epsilon}{\Delta Q}}$.

Definition 2.4 (Geometric Mechanism (GHOSH *et al.*, 2009))**.** Given any query $Q : \mathbb{N}^{|\mathcal{D}|} \to \mathbb{Z}^k$, the geometric mechanism defined as

Figure 9 – Geometric Mechanism overlapped with true values



Source: elaborated by the author.

$$\mathcal{A}_G(D, Q, \varepsilon) = Q(D) + (Y_1, \ldots, Y_k), \tag{2.2}$$

where Y_i are i.i.d. random variables drawn from $Geom(\frac{\varepsilon}{\Delta Q})$ and $\mathcal{D}$ is the set of all possible datasets, satisfies ϵ-DP with $\alpha = e^{-\frac{\varepsilon}{\Delta Q}}$.

Figure 9 shows an example of a non-perturbed output of a variable overlapped with the perturbed output of the geometric mechanism. The original output is represented by the blue bars, while the perturbed output is represented by red bars, and the space between them highlights the error introduced by the mechanism using $\varepsilon = 0.1$.

Definition 2.5 (Randomized Response Mechanism (DWORK *et al.*, 2006))**.** The RR mechanism stands out by allowing an input value v to be encoded into a bit vector B such that $B[v]$ may be represented by more than one bit assigned to 1. It was proved in (ERLINGSSON *et al.*, 2014) that the RR protocol satisfies ε-LDP if the bit vector B is perturbed according to probabilities $p = \frac{1}{1+e^{\varepsilon}}$ and $q = 1 - p$.

2.3.2 Differential Privacy Properties

Various useful properties are present in differentially private mechanisms, such as *post-processing*, *sequential composition*, and *parallel composition*. Combined, these properties offer the flexibility to aggregate multiple differentially private steps into a single mechanism that satisfies differential privacy.

These properties are formally stated below:

Theorem 2.6 (Post-processing (DWORK *et al.*, 2014)). Let $\mathcal{A}$ be any randomized algorithm such that $\mathcal{A}(D)$ is ε-differentially private, and let f be any function. Then, $f(\mathcal{A}(D))$ also satisfies ε-DP.

Theorem 2.7 (Sequential Composition (DWORK *et al.*, 2014)). Let $\mathcal{A}_i$ provide ε_i-differential privacy. A sequence of differentially private algorithms $\mathcal{A}_i(D)$ provides $\sum \varepsilon_i$-DP.

Theorem 2.8 (Parallel Composition (DWORK *et al.*, 2014)). Let each D_i be disjoint data and $\mathcal{A}$ an algorithm that provides ε_i-differential privacy for data D_i. A sequence of differentially private algorithm executions $\mathcal{A}(D_i)$ provides $\max(\varepsilon_i)$-DP.

2.3.3 Summary

In this chapter, we provide the theoretical foundations required to advance the discussion in the next chapters. We first describe the purpose and consent metadata, which is the main focus of this thesis. We then describe filters, which are the data structure we use to store the purpose and consent metadata. Finally, we describe differential privacy, which is a privacy-preserving mechanism that can be used to protect the purpose and consent metadata, focusing on the toolset that we will use in our solution, such as the geometric mechanism and the properties of differential privacy that we will use to ensure that our solution is differentially private. The composition theorems are used in our differential privacy proof for filter privacy, in Chapter 5, and the post-processing theorem is used to prove that the integration done in Chapter 6 is differentially private.

3 RELATED WORK

In this section, we review recent studies relevant to the key axes of our research: (I) the modeling and representation of purpose and consent, discussed in Section 3.1; (II) how filters can be exploited to reveal inserted elements with a degree of certainty, covered in Section 3.2; (III) other techniques for set membership using differential privacy; and (IV) strategies for ensuring privacy compliance in filters, detailed in Section 3.4. Finally, we summarize the findings in a comparative table and position our work in relation to the reviewed studies in Section 3.5.

3.1 Approaches for storing and representing consent data

Purpose-based access control has been extensively studied, with early works such as (BYUN; LI, 2008; AGRAWAL *et al.*, 2005; RIZVI *et al.*, 2004; AGRAWAL *et al.*, 2002; AGRAWAL *et al.*, 2005) proposing solutions that store purpose and consent metadata directly in database tables. More recent studies, driven by legal requirements introduced by regulations like GDPR, have explored alternative methods for modeling and storing purpose and consent metadata. For instance, works such as (SHASTRI *et al.*, 2020; PAPPACHAN *et al.*, 2020; DESHPANDE, 2021; KRASKA *et al.*, 2019) investigate approaches like surrogate keys or embedding metadata alongside data within the database.

Compliance by Construction (SCHWARZKOPF *et al.*, 2019) proposes storing personal data in user-specific shards, accessible only through materialized views, which may replicate data and increase storage requirements. SchengenDB (KRASKA *et al.*, 2019) proposes a method to store purpose metadata as bitmaps alongside tuples, suggesting Huffman encoding (HUFFMAN, 1952) to compress the metadata. GDPRBench (SHASTRI *et al.*, 2020) highlights the significant storage overhead of regulation metadata, showing that personal data storage can increase by up to six times, emphasizing the need for specialized solutions. Sypse (DESHPANDE, 2021) employs data partitioning, surrogate keys, and table-based storage for purpose metadata. Machado and Amora (MACHADO; AMORA, 2021) analyze the storage challenges posed by regulatory requirements and propose strategies to mitigate metadata explosion in GDPR-compliant databases. Konstantinidis et al. (KONSTANTINIDIS *et al.*, 2021) describe a middleware that uses first-order logic and constraints to perform query rewriting and enforce purposes, allowing for a fine-grained approach that gives more flexibility in choosing which data is consented (e.g.,

a tuple may not be included if two specific attributes are present in the query). Many of these works, as mentioned, focus on storing purpose metadata in restricted tables, requiring additional metadata construction layers. To provide a deeper understanding, we select three key studies (AGRAWAL *et al.*, 2002; PAPPACHAN *et al.*, 2020; SHASTRI *et al.*, 2020) for detailed analysis, emphasizing their approaches to modeling and storing purpose and consent metadata.

3.1.1 Hippocratic Databases (AGRAWAL et al., 2002)

Hippocratic Databases (AGRAWAL *et al.*, 2002) is an approach to database design centered on the principle of data privacy. The authors argue that as digital information becomes increasingly prevalent, database systems must fundamentally incorporate responsibility for the privacy of the data they manage. Drawing inspiration from the privacy tenet of the Hippocratic Oath, the paper lays out ten founding principles for these databases, derived from global privacy legislation and guidelines.

- **Purpose Specification:** Data should be collected for specific, legitimate purposes and not further processed in a manner incompatible with those purposes.
- **Consent:** Data should not be collected or processed without the informed consent of the data subject.
- **Limited Collection:** Only the data necessary for the specified purposes should be collected and processed.
- **Limited Use:** Data should not be used for purposes other than those for which it was collected.
- **Limited Disclosure:** Data should not be disclosed to third parties without the consent of the data subject.
- **Limited Retention:** Data should not be retained longer than necessary for the purposes for which it was collected.
- **Data Integrity:** Data should be accurate, complete, and kept up to date.
- **Security Safeguards:** Data should be protected by reasonable security safeguards against loss, unauthorized access, destruction, use, modification, or disclosure.
- **Openness:** Individuals should be able to access all their data.
- **Compliance:** Organizations should be accountable for complying with these principles and should have mechanisms in place to demonstrate compliance.

The paper includes a “strawman” design of a Hippocratic Database, which is a

relational database that incorporates these principles. The design includes the addition of a new attribute to each tuple, called the *purpose*, which specifies the purpose for which the data was collected. This attribute is used to enforce the purpose specification principle. The paper also discusses the challenges of implementing these principles in practice, such as the need for a new query language that can express purpose-based queries and the need for a new access control mechanism that can enforce the principles. The authors argue that Hippocratic Databases are a step towards more responsible and ethical data management practices, and they call for further research in this area.

3.1.2 Sieve (PAPPACHAN et al., 2020)

Sieve presents a middleware architecture designed to provide scalable fine-grained access control (FGAC) by separating access control policy enforcement from the core database engine. The main idea is to reduce the number of checked tuples and policies.

To reduce the number of tuples checked, Sieve employs Guarded Expressions, which are composed of a Guard, a low-cost predicate, and a Policy Partition, which is a subset of policies that apply to the Guard. These Guarded Expressions are used to create new indexes, which are used to evaluate the predicates.

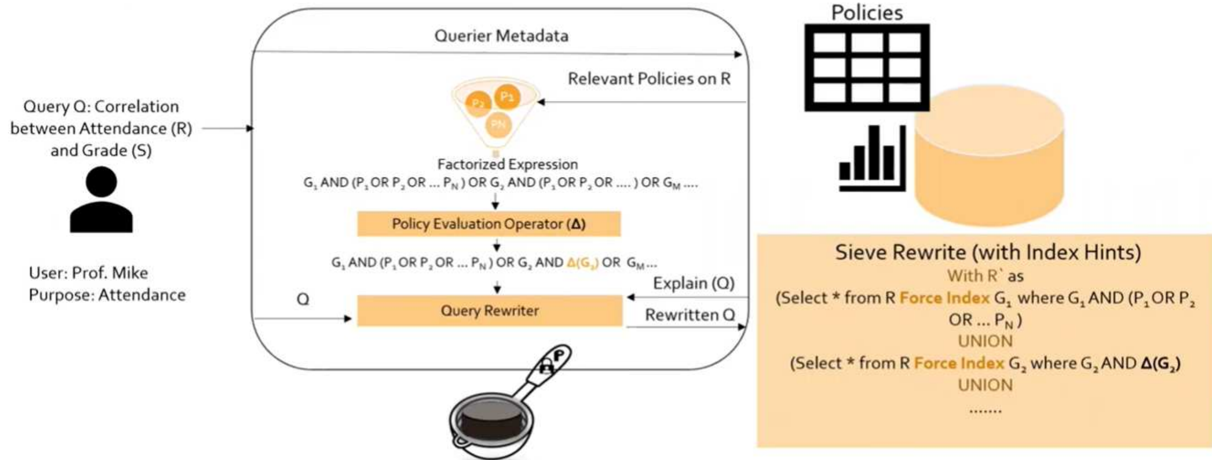
To reduce the number of policies checked, Sieve uses a Policy Evaluation Operator (Δ), which is a UDF that checks if a tuple satisfies a policy.

With these two tools, Sieve performs query rewriting, going through the following steps:

- **Initial Policy Filtering:** Filters the global set of policies, removing those that do not apply to the query.
- **Strategy Selection for Data Access:** Use the generated guards or a highly selective predicate as the primary indexed access method. This selection is made through cost estimation using the EXPLAIN operator.
- **Strategy Selection for Policy Evaluation:** For each guarded expression, Sieve selects either Δ or predicate inlining, based on the cost model.
- **Query Rewriting:** The query is rewritten using WITH clauses to include the selected guarded expressions and predicates, which are then used to access the data.

This query rewrite depends on some information to be stored in the database, as well as the creation of several indexes, which are used to optimize policy evaluation. The metadata

Figure 10 – Query workflow in Sieve



Source: (PAPPACHAN *et al.*, 2020)

includes mappings between users and policy identifiers, attribute-level constraints, permissible operations, and context-specific qualifiers (e.g., time, location, or device type).

Figure 10 shows the above process. In the example, we have the querier "Prof Mike" who will query the DBMS to know if there is a correlation between Attendance and Grades. To do so, he queries the DBMS with the purpose "Attendance", to collect data related to attendance in his classes. This metadata is checked against the "Policies" table, which will retrieve only the policies related to the "Attendance" table, and will provide an expression to be checked using the Policy Evaluation Operator (Δ) or using guarded expressions. Once the decision is made, the query is rewritten to include the necessary expressions, including the index hints, and will be executed against the database. The result will be a set of tuples that satisfy the query and the policies.

3.1.3 GDPRBench (SHASTRI et al., 2020)

GDPRBench analyzes and benchmarks the impact of complying with GDPR in an existing database by adding new attributes related to each right. The data model is described below:

- **Key:** A unique identifier to the object.
- **Data:** The personal data itself.
- **Metadata:** Seven attributes that directly reflect GDPR requirements. They are:
  - **Purposes (PUR):** A list of purposes for which the data may be used.
  - **Time-to-live (TTL):** The time until the data must be deleted.

- **User (USR):** The user to whom the data belongs.
- **Objections (OBJ):** A list of objections to the data processing.
- **Automated Decisions (DEC):** A flag indicating if the data may be used for automated decisions.
- **Third-party Sharing (SHR):** A flag indicating if the data may be shared with third parties.
- **Source (SRC):** The origin of the data, which may be a third-party or the user.

To provide a tuple example, we have:

```
KEY=ph-1x4b; DATA=123-456-7890;
PUR=ads,2fa; TTL=365days; USR=neo;
OBJ= ; DEC= ; SHR= ; SRC=first-party.
```

From the example, note that purposes are represented in plain text. OBJ, which can be our equivalent to consents, is represented as a set. One problem that GDPRBench highlights in their results is called Metadata Explosion, meaning that to comply with GDPR, the volume of added metadata may greatly surpass the volume of actual data, bringing attention to how to store this metadata better. Even if purposes are preconfigured in the database, this representation would result in at least a value representing each purpose, much like Hippocratic Databases (AGRAWAL *et al.*, 2002).

Discussion

From the works mentioned, purposes and consents are expressed and stored without concern for their value and the privacy of this data. While they are used as a component to enforce privacy rights to personal data, they themselves are not protected. We notice an opportunity to improve purpose and consent metadata representation, considering that the opt-in/opt-out may reveal unintended information about members. Notably, the problem of consent enforcement can be seen as a set intersection problem, as described in Section 2. This inspired us to use access filters. By using filters, we leverage their access and storage capabilities with added uncertainty through the probabilistic nature. However, filters bring a significant problem to consent enforcement: allowing false positives means that data from a person who opted out (negative) would be returned (false positive). This brings us to our first milestone: designing a filter that does not have false positives, respecting consents. We compare our work with both

Agrawal et al.(AGRAWAL *et al.*, 2002) and Sieve (PAPPACHAN *et al.*, 2020), to evaluate what is our added overhead. As shown in Chapter 7, we do not add significant overhead to storage.

3.2 Attacks on filters

As mentioned in Section 2, filters are a probabilistic data structure. Therefore, they are expected to have some measure of privacy on the elements used to construct a filter. However, works like BLIP (ALAGGAN *et al.*, 2012) and Bianchi et al. (BIANCHI *et al.*, 2012) investigate attacks that make it possible to re-identify the elements previously inserted in a Bloom Filter. More recently, Reviriego et al. (REVIRIEGO *et al.*, 2023a; REVIRIEGO *et al.*, 2024; REVIRIEGO *et al.*, 2023b) conducted a more comprehensive study on the vulnerabilities of other types of filters, such as Cuckoo Filters (FAN *et al.*, 2014), Quotient Filters (PANDEY *et al.*, 2017), and Counting Bloom Filters (BONOMI *et al.*, 2006). We shall focus on the attack on Counting Bloom Filters since it is more aligned with our goal.

Reviriego et al. show in (REVIRIEGO *et al.*, 2023a; GALÁN *et al.*, 2023) that Counting Bloom Filters can be attacked with both access to the filter itself (white box) and access only to the interface, with operations like insert, query, and delete (black box). Both works rely on a peeling algorithm, which will extract elements from the filter one at a time as if peeling layers from an onion. From the white box version, the algorithm can be described as:

1. Create a set P by testing all the universe U against the filter, storing the positives (true or false) in P.
2. P can be observed as an union of S, which are the true positives, and F, false positives. The goal is to extract S from P.
3. Construct a hash table T using the same hash functions $h_1(x), h_2(x), ..., h_k(x)$ used in the Counting Bloom Filter and insert all elements in P into those positions $T[h_1(x)], ..., T[h_k(x)]$
4. Search the hash table for a position that stores the same number of elements as the counter value. For example, consider the case when the counter has a value of 1, and there is only one element, z. If such a position is not found, the procedure ends with a failure. If such a position is found, the elements mapped to it must be elements of S because only those could have incremented the counter. At this point, we can add those elements to our recovered set S_{rec}.
5. Remove the elements added to S_{rec} from the k positions they map to in the hash table and decrement the counters on those positions in the Counting Bloom Filter. If any counter

reaches a value of 0 as a result, then we check the hash table; if there are elements in the corresponding position of the counter, then we can remove them as they are false positives.

6. If there are still elements in the hash table, search again for a position that stores the same number of elements as the counter value and repeat the process. If the hash table is empty, then $S_{rec} = S$, and we have successfully recovered the set from the filter.

Figure 11 – Process of extraction of true positives from a Counting Bloom Filter

Elements inserted: {x,y,z}   False Positives: {e,g,h}

		Remove X		Remove E		Remove Z	
0		0		0		0	
0		0		0		0	
2	{x,z,e}	2	{x,z,e}	1	{z,e}	1	{z}
0		0		0		0	
1	{x,e}	1	{x,e}	0	{e}	0	
1	{y,g}	1	{y,g}	1	{y,g}	1	{y,g}
1	{z,h}	1	{z,h}	1	{z,h}	1	{z,h}
0		0		0		0	
1	{y,g,h}	1	{y,g,h}	1	{y,g,h}	1	{y,g,h}
1	{z,e}	1	{z,e}	1	{z,e}	1	{z}
1	{x}	1	{x}	0		0	
1	{y,g,h}	1	{y,g,h}	1	{y,g,h}	1	{y,g,h}

...

Source: Adapted from (REVIRIEGO *et al.*, 2023b)

Figure 11 shows this process. In this example, we have 3 elements inserted in a Counting Bloom Filter, $\{x, y, z\}$, and three false positives $\{e, g, h\}$. The first step is to create the hash table T with the elements in the filter, which will be used to recover the true positives. The second step is to find a position in the hash table that has a counter value of 1, which means that there is only one element mapped to it. In this case, we find $h_3(x)$. Since we found a position with one element and a counter value of 1, we can add x to the recovered set S_{rec}. The next step is to remove x from the hash table and decrement the counters. Now we find a position with one

element (e), however, the counter on the filter is 0, therefore, e can be safely discarded as a false positive. The process continues, now for z, and goes until the hash table is empty, and we have recovered all true positives.

There can be collisions (e.g., $h_i(x) = h_j(x)$), so an element may be mapped twice in the hash table T, so the element is added as many times as it maps, but it is only added to S_{rec} once. When removing it from T, all copies are removed.

This method also has a constraint related to the false positive set size. The method only works if the false positive set and the true positive set are of about the same order of the number of elements. Nevertheless, it is uncommon to use a filter with a false positive rate higher than, say, 10%.

3.3 Privacy on set membership data

As mentioned in Chapter 2, consent enforcement can be seen as a set membership problem, where the goal is to check if an element is in a set. In our research, we also found other data representations that can be used to perform set membership queries. We highlight three works, discussing how they do not exactly fit with our problem definition.

3.3.1 Private Set Intersection (FREEDMAN et al., 2004)

Private Set Intersection (PSI) is a strategy in which encrypted sets are compared to find the intersection between them. The core method ensures that one party (client) learns only the intersection of the two sets, while the other party (server) learns nothing about the other set. This is done through homomorphic encryption and a polynomial representation of the sets.

The steps are:

- The client constructs a polynomial in which each root is an element of their set.
- The client encrypts the polynomial coefficients using a homomorphic encryption scheme, sending the result to the server.
- For each element in the server's set, it uses the homomorphic properties to evaluate the polynomial at that element, without learning the coefficients or the polynomial itself.
- The server masks the results before sending them back to the client, ensuring that the client only learns which elements are in the intersection.
- The client decrypts the answer, learning which elements were in the intersection, and

gibberish for the rest.

As mentioned before, this is not exactly aligned with our problem definition, as the cryptography scenario incurs in total utility loss, which is not desirable in our work. We move to the next work, which uses differential privacy.

3.3.2 Differentially Private Set Intersection and Union (XUE et al., 2017)

Xue et al. describe an algorithm to perform differentially private set intersection and union. The algorithm is based on the concept of randomized response and applied to local differential privacy. The main idea is to get perturbed bit arrays from users, and, by knowing the odds of bit-flipping, which are public in the protocol, the collector attempts to reconstruct the original intersection of the sets.

Users encode their data in bit arrays, where each bit represents an element in the universe. The algorithm gets the perturbed arrays, computes the number of ones on each column, and, using maximum likelihood estimation, through the probabilities of flipping, it estimates the original matrix. To enhance utility, the variance of the estimate is calculated, and an element is considered in the intersection if the error is below this threshold.

This approach relates to profile building and local-DP, not being compatible with our set membership problem.

3.4 Approaches for private filters

The lack of privacy in filters motivated several works to provide some type of privacy guarantee in filters. While some approaches will add noise to the filter, others will mostly manipulate the set of elements inserted in the filter in order to generate enough uncertainty. This distinction is important because added noise to the filter means that it will lose the important property of no false negatives, while adding dummy elements may largely increase the false positive rate. Nevertheless, we must be careful not to completely destroy the usefulness of the filter.

First, we describe the work of Bianchi et. al. (BIANCHI *et al.*, 2012), which proposes γ-deniability, which is an extension of K-anonymity for Bloom Filters.

Secondly, we describe BLIP (ALAGGAN *et al.*, 2012), an ε-DP algorithm for Bloom Filters, which uses the randomized response mechanism to ensure differential privacy

Finally, we describe DPBloomFilter (KE *et al.*, 2025), an (ε, δ)-DP algorithm, also for Bloom filters, that uses randomized response; however, the parameters are different.

3.4.1 Bianchi et. al. (BIANCHI et al., 2012)

The work describes privacy metrics for Bloom Filters, and, based on the concept of hiding set, proposes a γ-k-anonymity algorithm and a special case γ-2-anonymity called γ-deniability.

The intuition for the privacy metrics and analysis is that an element may cast doubt over its presence in the filter by claiming a false positive result, hence, deniability.

The hiding set is a set of dummy elements which for all of these elements, the filter returns positive. The strategy of γ-deniability takes advantage of this hiding set to ensure deniability for the elements.

An element x inserted in a Bloom Filter is deniable if for all hash functions, there is at least one element υ from the hiding set such that $h_i(x) = h_j(\upsilon)$. A Bloom Filter is γ-deniable whenever a randomly chosen x is deniable with probability γ.

Figure 12 shows an example of a 0.66-deniable filter. We have the real elements $\{x1, x2, x3\}$ and the dummy elements $\{\upsilon1, \upsilon2, \upsilon3\}$, that form the hiding set and are specifically inserted to hit false positives for the real elements. Note that no position for each real element $x1, x2$ can be truly asserted because the hiding set has elements in the same positions. But for $x3$, there is no element in the hiding set covering position $B[11]$, which is why the value of γ is $\frac{2}{3}$, or 0.66.

3.4.2 BLIP (ALAGGAN et al., 2012)

BLIP (BLoom-then-flIP) is a ε-differential privacy mechanism to randomize Bloom Filters. The namesake also describes the process, in which a Bloom Filter is constructed, then, each bit of the Bloom Filter is passed through a Randomized Response mechanism, flipping the bits with probability $\frac{1}{1+e^{\frac{\varepsilon}{k}}}$. Figure 13 shows a visualization of the process on a small scale, a filter with 18 bits, 3 hash functions, and 3 elements inserted:

For each bit, the algorithm flips it with probability $\frac{1}{1+e^{\frac{\varepsilon}{k}}}$; in the figure, they are represented by the orange positions. This bit flipping inserts the false negative error, and may increase the false positives, as we observe in positions $B[1]$ and $B[12]$. $B[1]$ being flipped makes b return negative, when it was inserted in the filter and should return positive, making it a false

Figure 12 – Bloom filter composed of elements $x1, x2, x3$ that admits 3 false positives: $v1, v2, v3$.

X_1	1	0	1	0	0	0	0	0	0	0	1	0
X_2	0	0	1	0	1	0	0	0	0	0	1	0
X_3	0	0	0	0	1	0	0	1	0	0	0	1
BF	1	0	1	0	1	0	0	1	0	0	1	1
V_1	1	0	1	0	1	0	0	0	0	0	0	0
V_2	0	0	1	0	0	0	0	1	0	0	1	0
V_3	1	0	0	0	0	0	0	1	0	0	1	0

Source: Adapted from (BIANCHI *et al.*, 2012)

negative. $B[12]$, in turn, adds w to the positive results when it is not inserted in the filter; hence, it is a false positive. From this example, we can see the impact that adding differential privacy through bit-flipping can cause in filters.

The work also describes a profile reconstruction attack based on the use of Bloom Filters as encoding data structures for user profiles in social media. The attack goes as follows:

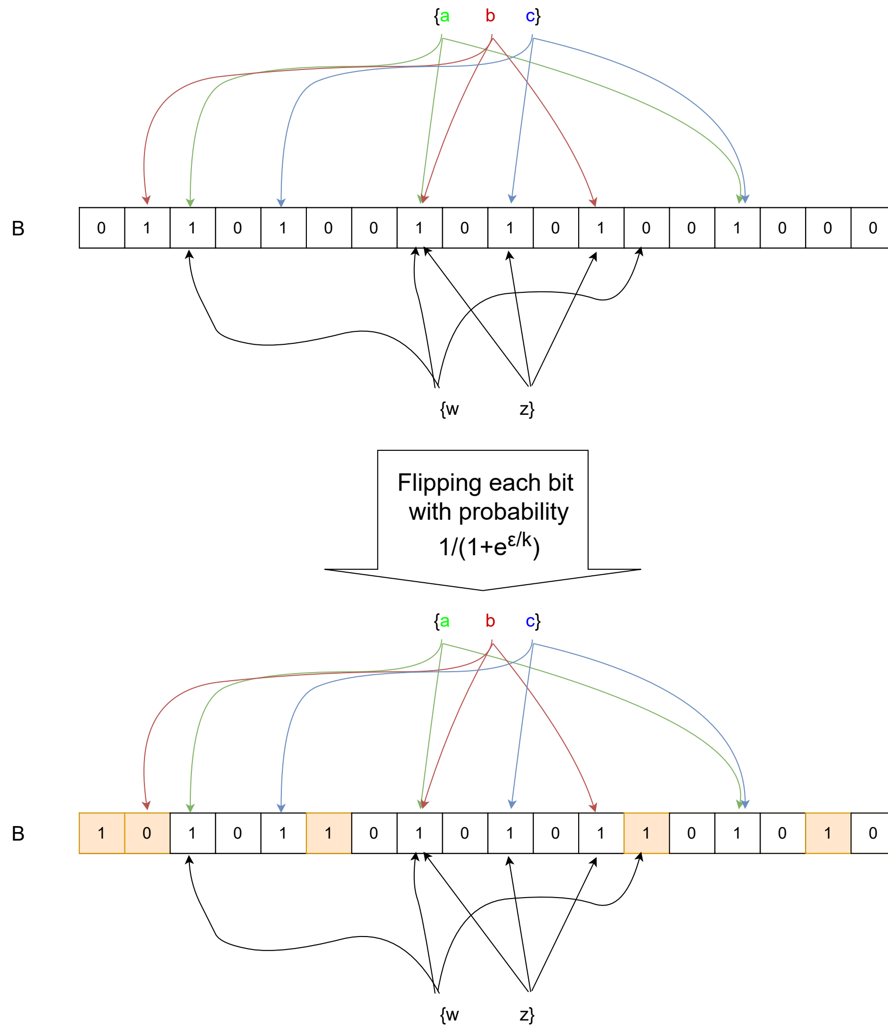
- For each item x in the universe, the adversary checks the results of each $h_i(x)$, setting k_0 for the 0-valued bits and k_1 for the 1-valued bits.
- Then, a calculation is made through the expression $p^{k_1}(1-p)^{k_0}\binom{k_1+k_0}{k_0} > c$, where c is a constant between 0 and 1.
- If the condition is true, the item is added to the reconstructed profile.
- Finally, the reconstructed profile is compared to the original in terms of cosine similarity.

3.4.3 DPBloomFilter (KE et al., 2025)

DPBloomFilter provides (ε, δ)-differential privacy for membership queries in Bloom Filters using a Randomized Response mechanism. The algorithm and takeaways are similar to BLIP, however, instead of flipping the bits using $\frac{1}{1+e^{\frac{\varepsilon}{k}}}$, it uses $\frac{1}{1+e^{\frac{\varepsilon}{N}}}$, where N is a random variable sampled from the $(1-\delta)$-quantile of a distribution W generated from the parameters of the Bloom Filter.

This variable is sampled from the distribution of W, shown in Figure 14, which is the set of positions where two neighboring datasets differ. This is calculated through the distribution

Figure 13 – Bloom filter before and after BLIP algorithm.



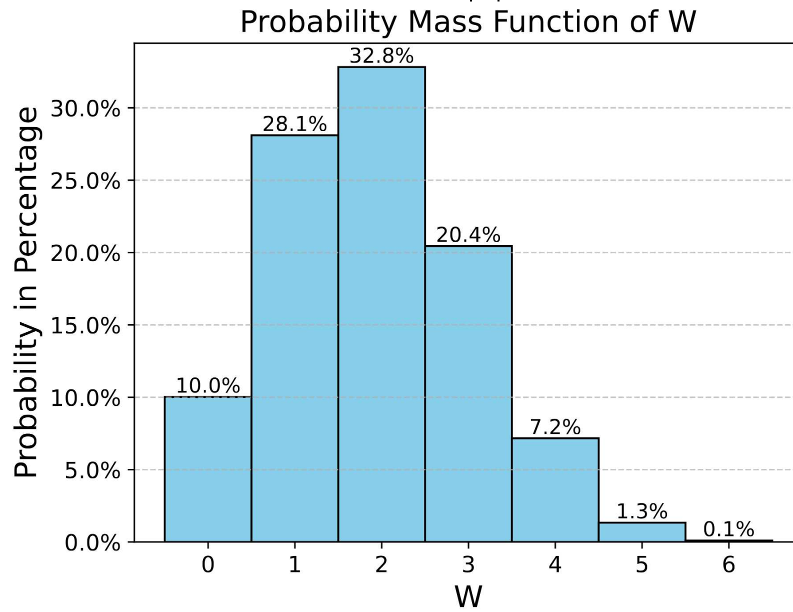
Source: elaborated by the author.

of Y_x, which is the set of distinct values generated by the hash functions for a given value x, and Z, which unions two different Y sets, Y_x and $Y_{x'}$.

Figure 14 shows the distribution of W for a Bloom Filter with $k = 3$, $m = 2^{19}$ and $|P| = 10000$. N is the $(1-\delta)$-quantile of this distribution, which is used to calculate the probability of flipping the bits in the Bloom Filter, ensuring (ε, δ)-DP. The distribution shows good concentration properties around its mean.

The work also provides privacy guarantees, proving that the added noise is sufficient for differential privacy and utility analyses, comparing DPBloomFilter to a non-perturbed Bloom filter.

Figure 14 – Distribution W for $k = 3$, $m = 2^{19}$ and $|P| = 10000$



Source: (KE *et al.*, 2025)

3.4.4 Discussion

Privacy in filters is a renewed field of study. While works like Bianchi et. al. (BIANCHI *et al.*, 2012), RAPPOR (ERLINGSSON *et al.*, 2014) and BLIP (ALAGGAN *et al.*, 2012) raised the problem of privacy of elements in a filter, and suggested solutions, they focus only on Bloom Filters and as a profile representation, aiming for similarity between two filters, not using the filter as a set membership problem. Reviriego et. al. (REVIRIEGO *et al.*, 2023a; REVIRIEGO *et al.*, 2024; REVIRIEGO *et al.*, 2023b) recently conducted a vulnerability study on several types of filters, providing opportunities to research and amend those vulnerabilities. Differential privacy can offer tight guarantees and allow for the utility of those data representations, with works like DPBloomFilter(KE *et al.*, 2025) exploring DP for Bloom Filters in a set membership problem. Our experimental evaluation, presented in chapter 7, follows the analyses of DPBloomFilter, using it as a baseline whenever applicable, showing the evolution of the false positives and false negatives with different filter configurations.

3.5 Summary

After detailing a variety of related works to our solution, both in consent and purpose metadata representation and in differential privacy applied to a specific data structure, we finish this section by presenting Table 1, which summarizes these existing works. It compares the

consent representation, where this data is stored, if there are any privacy concerns over the consent data, and what the associated privacy mechanism is.

Work	**Consent representation**	**Consent enforcement**	**Privacy concerns**	**Mechanism**
(AGRAWAL *et al.*, 2002) (VLDB)	Added attribute	Added predicate	N/A	N/A
(PAPPACHAN *et al.*, 2020) (VLDB)	Added tables	Middleware	N/A	N/A
(SHASTRI *et al.*, 2020) (VLDB)	Added attribute	Stored procedure	N/A	N/A
(BIANCHI *et al.*, 2012) (PSD)	N/A	N/A	Yes	γ-deniability
(ALAGGAN *et al.*, 2012) (SSS)	N/A	N/A	Yes	RR
(KE *et al.*, 2025) (Preprint)	N/A	N/A	Yes	RR
Our work	Purpose Filter	Embedded	Yes	Geometric

Table 1 – Summary of the existing works.

Several studies (AGRAWAL *et al.*, 2002; PAPPACHAN *et al.*, 2020; SHASTRI *et al.*, 2020) have been made to comply with consent enforcement, but with no special consideration for the stored data. Moreover, none of them consider the semantics of consent metadata, which, depending on the application and knowledge of an attacker, may reveal unintended information about the existence of a given element in a set (e.g., a person’s authorization to share data). Works like (REVIRIEGO *et al.*, 2023a; REVIRIEGO *et al.*, 2024; REVIRIEGO *et al.*, 2023b) show that filters are vulnerable to reconstruction attacks, meaning that a person’s consent to share data may be discovered. Other filter works (BIANCHI *et al.*, 2012; ALAGGAN *et al.*, 2012; KE *et al.*, 2025) focus on protecting the filter; however, the noise added results in false positives and false negatives, which hinders the usage of these filters to represent consent. To counter this type of attack, our work provides differential privacy guarantees for plausible deniability in this set. Our work proposes to store and protect consent metadata with formal guarantees, avoiding attacks and respecting consent constraints, unifying the two research fronts presented, appropriate consent metadata representation, and concern for this type of data and its privacy.

4 PURPOSE FILTERS

To store and query purpose metadata, we propose a filter-based data structure, named Purpose Filter. Purpose Filter is composed of multiple filters layered upon one another, each layer responsible for a different type of verification. A design inspired by Cascaded Bloom Filters (TRIPUNITARA; CARBUNAR, 2009) and Stacked Filters (DEEDS *et al.*, 2021), with the distinct objective of succinctly representing a data set and querying purpose data without violating purpose properties and definitions.

Each purpose is created as a Purpose Filter, and all positive or negative consents pertaining to it are used in filter construction. We build the *allow* and *deny* sets using the tuple ID or any unique identifier of this register in the database, such as an OID. Being a succinct structure, we store a representation of this tuple ID within the filter through hashing.

Whenever a query attempts to read personal data from a table, before retrieving this data from the disk, the tuple ids are retrieved from the DBMS Catalog, before data is actually scanned. The IDs are compared against the corresponding filter, and only the accepted tuples are returned. This verification can be cheap when the filter is in memory, since it only costs an access to the filter to ensure tuple acceptance. Details about this cost are provided in section 4.6. Due to the constraint of only returning allowed tuples, we cannot allow a false-positive tuple.

Purpose Filter layers can be of any compatible type of filter. We define "compatible" by having the same primitive behavior as a Bloom Filter (BLOOM, 1970). Therefore, Cuckoo Filters (FAN *et al.*, 2014), Counting Quotient Filters (PANDEY *et al.*, 2017), and XOR Filters (GRAF; LEMIRE, 2019) are examples of compatible filters. In this work, we focus on the construction of the Purpose Filter using Bloom Filter layers.

4.1 Filter Construction

Purpose Filter is built by layering filters one on top of the other, alternating the layers between positive and negative. The final data structure may have more than one pair of filters; however, it will always have an even number of layers. Algorithm 1 provides more detail.

The construction takes two sets, one positive of known elements in the set and one negative of known elements not in the set. The positive set is the "allow" set. The negative set is the complement of the "allow" set, i.e., the "deny" set. We also set a target false negative rate (FNR), which we use as a criterion to stop the construction process. Lines 3 to 6 show what

happens during construction: two layers are created per iteration, the positive set is inserted in the first layer, and the negative set is tested against this layer (lines 3 and 4). Then, any accepted elements from the negative set are inserted in the second layer, and the positive set is tested (lines 5 and 6). This process repeats until a target FNR is reached or it cannot be reduced further. Since the size of the sets diminishes over the iterations, subsequent layers are smaller, and the final data structure may have the form of a funnel.

Algorithm 1: Purpose Filter Constructor function

```
   Data: Positive set P, Negative set N, False Positive Rate FPR, # of hashes k
   Result: Constructed Purpose Filter
 1 filter = ;
 2 finished = false;
 3 filter.layers = {};
 4 while finished == false do
 5     R_set = {} // Elements that survive probing
 6     if filter.layer_counter % 2 == 0 then
 7         build layer with appropriate size ;
 8         for i=0 to P.length do
 9             layer.insert(P[i]);
10         end
11         for j=0 to N.length do
12             if layer.probe(N[j]) == true then
13                 R_set.add(N[j])
14             end
15         end
16         N = R_set // Update negative elements to only survivors
17         filter.layers.add(layer);
18     end
19     else
20         build layer with appropriate size;
21         for i=0 to N.length do
22             layer.insert(N[i]);
23         end
24         for j=0 to P.length do
25             if layer.probe(P[j]) == true then
26                 R_set.add(P[j])
27             end
28         end
29         P = R_set // Update positive elements to only survivors
30         filter.layers.add(layer);
31         finished = FNR_is_ok();
32     end
33 end
34 return filter;
```

In Algorithm 1, lines 7 and 20 refer to an "appropriate size". When building the Purpose Filter, we have two correlated variables, the number of hashes k and the layer FPR. If the number of hashes is passed as a parameter, the layer is constructed with a predefined size being a proportion of the positive set, yielding a varying FPR. If FPR is passed, the size of the first layer and the number of hashes are estimated using the positive set size; then these calculated variables are used as parameters for the construction. On line 31, a function rechecks the full filter False Negative Rate. If it is below a given threshold (e.g., the target FPR of the first layer) or has not changed, the filter is considered complete.

The size of each layer is calculated in section 4.5, being dependent on the type of filter used in every layer. We perform the calculation for Bloom Filters, and we show that on average, the biggest layer is the first layer, dominating the cost.

Our main goal is to achieve a zero false positive rate (0 FPR). From our consent modeling, we have a constraint that no data should be provided when the user does not give consent. We can derive that while no false positives may occur, a false negative is acceptable, because it does not violate the constraint. Regular filters may reduce the false positive rate (FPR), but do not guarantee a 0 FPR. A false positive defeats the system's compliance; therefore, we ensure that we have a 0 FPR.

We achieve the 0 FPR through Theorem 4.1. To start, we provide two claims, which must be true for the Purpose Filter.

Claim 5. A tuple z is deemed positive if rejected by a negative layer of the filter

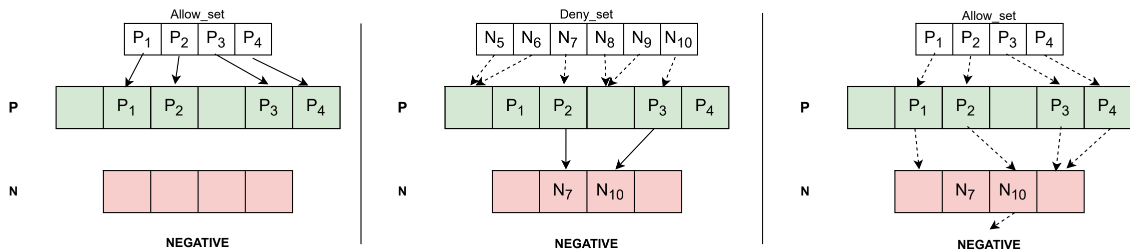
Claim 6. A tuple z is deemed negative if rejected by a positive layer of the filter or reaches the end of the filter.

Theorem 4.1. Let PF be a Purpose Filter; D the complete dataset; P, N, U distinct subsets of D such that $D = P \bigcup N \bigcup U$. P are the positive elements used in construction, N are the negative elements used in construction, and U are the undefined, which are elements of D not used in construction. If $U = \varnothing$ and the end of the filter returns negative, we achieve a 0 FPR.

Proof. Take elements $x, y \in D$. If $U = \varnothing$, be $x \in P$ and $y \in N$. During filter construction, x is inserted in the top layer and either is rejected by the next negative layer, directly below, or is inserted two layers below, in the next positive layer; y is either rejected by the topmost layer or inserted in the next negative layer. After construction, when we probe the filter for x, x either is rejected by a negative layer (true positive) or reaches the end of the filter (false negative); when we probe for y, y is either rejected by a positive layer (true negative) or reaches the end of the filter (true negative). Since any element in D is only part of P or N, it is not possible to have a false positive. □

Through Theorem 4.1, we prove that there is a design for a filter structure that represents purpose and stores consent metadata, complying with these data constraints of avoiding a false positive.

Figure 15 – Purpose Filter construction.



Source: elaborated by the author.

To show the construction process, we go to our example. Figure 15 presents three phases of the building structure: the leftmost figure represents the beginning of construction, where the first layer is populated by the *Allow_set*. We already show the negative layer for clarity, but as Algorithm 1 shows, the layers are built incrementally. In the center figure, we have the second step, which is to get the *Deny_set* and test it against the positive layer. As the dashed arrows show, $\{N_5, N_6, N_8, N_9\}$ do not survive the first filter; therefore, they can be safely discarded as true negatives. The survivors $\{N_7, N_{10}\}$ are added to the negative layer, represented by the solid arrows. Then, in the rightmost figure, we return to the *Allow_set*, to verify how they

behave when checked by the negative layer, as we can see from the dashed arrows, $\{P_1, P_3, P_4\}$ are rejected by the negative layer, meaning that they are true positives. $\{P_2\}$ goes through the negative layer; however, the end of the filter dictates that any element that passes through every layer will be deemed negative.

The proper steps to guarantee Theorem 4.1 are taken during Purpose Filter construction and data probing. The positive set is built with all the relevant elements. All other relation tuples are added as the negative set to guarantee that all elements are used. That ensures the first condition, of $U = \varnothing$. All elements are known during filter construction because they are all tuples in the relation.

4.2 Filter Probing

After construction, the filter uses a probing function to respect both definitions used in Theorem 4.1, which algorithm 2 shows. That ensures the second condition, i.e., the filter returns negative at the end.

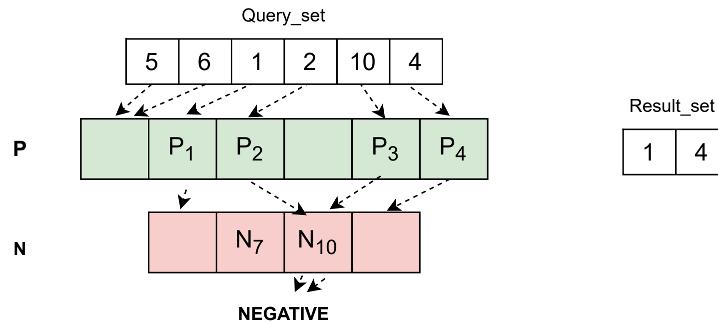
Algorithm 2: Purpose Filter probing function

```
   Data: Tuple x, Purpose Filter pf
   Result: true, if tuple is definitely in set; false, otherwise
1  for i = 0 to pf.layers.length-1 do
2  |  if pf.layers[i].lookup(x) == false then
3  |  |  return ((i mod 2) == 1);
4  |  end
5  end
6  return false;
```

To verify consent for a given tuple, the filter tests whether the tuple is accepted or rejected for each layer. If it is accepted, the algorithm moves to the next layer. If rejected, the filter evaluates on which layer it was rejected. If it is a positive layer (even-numbered layers), the filter returns negative. If it is a negative layer (odd-numbered layers), the filter returns positive. If the algorithm reaches the end of the filter, it returns negative.

The cost to probe is, in the worst case, the cost to check each layer. During the construction process, we verify that not too many layers are constructed after the first pair. For 10 million IDs used in our experiments, the filter had 4 layers. Therefore, it is still a fast check. We provide further analysis in subsection 4.6.

Figure 16 – Probe example



Source: elaborated by the author.

Figure 16 shows an example of a query posed to the DBMS, where we shall assume that the predicates would return tuple ids $\{1, 2, 4, 5, 6, 10\}$. From our running example, we know that from the intended result set, only $\{1, 2, 4\}$ consented to their data to be returned. The filter ensures that only consented tuple IDs return, filtering out tuple IDs $\{5, 6\}$ on the first layer. Tuple ID $\{10\}$ is accepted by the first layer; however, since it was added in the second layer, it reaches the end of the filter and is filtered out. Positive tuple ids $\{1, 4\}$ are rejected by the second layer, being added to the result set. Positive tuple ID $\{2\}$, though, collides with negative tuple ID $\{10\}$, and is accepted by the negative layer, being discarded. Finally, the result_set is built with the surviving tuple ids $\{1, 4\}$, and $\{2\}$ was discarded as a false negative.

With these changes, we achieve a purpose representation that is smaller in size than a usual solution, e.g., store the purposes as a new attribute (SHASTRI *et al.*, 2020; AGRAWAL *et al.*, 2002). Purpose Filter also complies with the requirement of not allowing any registers outside the purpose to be wrongly returned. Another benefit is that by embedding the Purpose Filter into the operator execution, no query rewriting or additional statements are required. All the logic is executed within query processing, avoiding processing overhead when executing queries, and providing an extra security layer for SQL malicious manipulation.

The collateral effect is to have a false negative rate. That is inherent to the data representation performed by the filter. However, purpose-aware query processing is respected. We evaluate how this false negative rate evolves over data proportions of positive and negative, and conclude that the number is small enough, most often near zero. More details, including the modeling of this false negative rate, are available in subsection 4.4.

4.3 Filter Maintenance

In the case that new personal data is added, the Purpose Filter must be updated before any probing of new data, since we are restricted by Theorem 4.1 that the unknown set must be empty. As consent can be allowed or denied, this new element will be part of the positive or negative set.

If the element is part of the positive set, it must be inserted into the top layer of the filter, which is positive. This operation is enough to ensure the consistency of the filter. However, to dampen the growth of the FNR, this element can follow through the layers of the filter, being tested against the next layer (negative). If it yields a false negative, it will be added to the next positive layer until it reaches the end of the filter. This gives extra chances for the element to be correctly classified as a true positive.

If the element is part of the negative set, it must be inserted in the second topmost layer of the filter, which is the first negative layer of the filter. Unlike the previous decision, the negative element must be inserted in every negative layer until it reaches the end of the filter. The reason behind this is that if there is a negative layer that does not "see" this element, it may reject it, yielding a false positive, which is unacceptable.

Bloom filters, by definition, do not support deletes, but other structures such as Cuckoo Filters (FAN *et al.*, 2014) and Counting Quotient Filters (PANDEY *et al.*, 2017) do support deletion, and are compatible as layers of Purpose Filter, as long as all are the same. Modification of a consent can be seen as a delete-and-insert operation. Inserts were already covered; let us focus on deletes.

To delete a positive entry, it is only necessary to remove it from the topmost layer of the filter. This may increase the FNR because other entries may be affected because of the removal of a given entry (e.g., hash collisions).

An issue arises when deleting negative entries because they may incur false positives in the negative layers. Currently, this is a limitation, although it is more probable that a user will revoke consent after granting it instead of the contrary.

4.4 Information Loss

The foundation of Purpose Filter, and purpose-based access control in general, is to restrict unwarranted access to all data, allowing only authorized tuples to be returned as a query

result. Therefore, within the same dataset, a given query Q posed to a compliant DBMS may return fewer tuples than the same query in a noncompliant DBMS. Consequently, it is important to model this information loss accordingly. We will use FNR for the false negative rate of the filter and FPR_i for the false positive rate of the i-th layer. To clarify, FNR applies to the entire filter, while FPR_i refers to each layer.

The possible outputs are: True positive, True negative, False Negative. Since we are interested in false negatives, the probability of a positive element reaching the end of the filter is the same as that of its being accepted by every layer. So we must associate the probabilities of it being accepted in the positive layers (half of the filter) and the negative layers (the other half).

Since the element is positive, the probability that it is accepted by a positive layer is 1, therefore, we can ignore the even layers in the FNR calculation:

$$TotalFNR = \prod_{i=0}^{\frac{\#_L}{2}} FPR_L_{2i+1} \tag{4.1}$$

4.5 Space allocation

As expected, the allocation of space for filter layers depends on the set size that will be inserted. For the first layer, *nPos* is the positive set size and *nNeg* the negative set; however, for each subsequent layer, *nPos* and *nNeg* correspond only to the survivors. We have positive survivors, which yielded a pass when they should have been rejected from a negative layer, and negative survivors, which have the same behavior from a positive layer. The expected size of the positive survivor set at a layer *i* is:

$$nPos_i = nPos * FPR_i * \prod_{j=0}^{i} FPR_{2j}$$

Similarly, the expected size of the negative survivor set at a layer *i* is:

$$nNeg_i = nNeg * FPR_{i+1} * \prod_{j=1}^{i} FPR_{2j-1}$$

Summing all the Bloom Filter layers, we obtain:

$$TotalSize = \sum_{i=0}^{\frac{\#_L}{2}} \frac{(nPos_{2i} * \log p_{2i})}{(\log 2)^2} + \frac{(nNeg_{2i+1} * \log p_{2i+1})}{(\log 2)^2} \tag{4.2}$$

By examining the productories, we can observe that the survivor set size diminishes very fast, since we have a product of very small values (the FPRs).

4.6 Costs

To quantify the costs of the purpose filter, we establish two constants c_{ins} and c_{prb}, which represent the cost of inserting an element into a layer and the cost of detecting the layer for the element. For Bloom Filters, $c_{ins} = c_{prb} = k$, since the operations are bound for the k hash functions used.

Building Cost To build each layer, we insert every element of the corresponding set and then probe it using the opposite set. For a positive layer, we have:

$$c_{ins} * (nPos_i) + c_{prb} * (nNeg_i)$$

And for a negative layer:

$$c_{ins} * (nNeg_i) + c_{prb} * (nPos_i)$$

The resulting build cost is the sum of these costs for each layer

$$BuildCost = \sum_{i=0}^{\frac{\#_L}{2}} c_{ins} * (nPos_{2i}) + c_{prb} * (nNeg_{2i}) + c_{ins} * (nNeg_{2i+1}) + c_{prb} * (nPos_{2i+1}) \tag{4.3}$$

Observe that in the last pair of layers, the term $c_{prb} * (nPos_{2i+1})$ does not exist.

The time complexity for construction is bound by $O(k * (nPos + p * nNeg) * l)$, where k is the number of hash functions, nPos and nNeg are the sizes of the positive and negative sets, p is the FPR of the layer, and l is the number of layers.

Query Cost The query cost is either the sum of the rejection probability of each layer or the probability that all layers accept it. For positive and negative elements, the cost is bound by:

$$QueryCost \leq \max\{\sum_{i=0}^{\frac{\#_L}{2}} c_{prb} * (1 - FRR_i), \prod_{i=0}^{\frac{\#_L}{2}} c_{prb} * FRR_i\} \tag{4.4}$$

Therefore, the time complexity is bound by $O(kl)$ for each element.

4.7 Optimizing filter construction

We can fine-tune the construction process by making use of three constraints: space, target FPR_0, and target FNR. If there are space limitations, the FNR may vary because we need to adjust the number of bits per element. Do note that the build threshold may impact the amount

of subsequent layers, but in general, as already shown, the bulk of storage space is used by the first layer. By affixing FPR_0, we do not hinder the total FNR of the structure, as the experiments will show.

4.8 Purpose and Consent Modeling using Purpose Filter

Consent may be given as a blanket term (i.e., the owner consents to the usage of all their data or denies all). For each purpose, a filter is created, and each user's consent (allow/deny) is added to the filter. In this scenario, only one filter per purpose is needed, as we only have one consent per tuple. Consent can also be given as a fine-grained collection of consents for each attribute of the tuple. This is called cell-level consent (KONSTANTINIDIS *et al.*, 2021). For cell-level consent, we build a filter for each attribute and each purpose. The consents for a given purpose are presented as a list of allow/deny for each attribute, which is why all consents for a given attribute are applied to a purpose filter.

For example, let us consider a table T with the following attributes: A, B, C, and D and 2 users, X and Y, that have their data in T. A purpose P is created in table T, for tuple-level consent, meaning that one Purpose Filter will be associated with T. X and Y provide their consents, where X agrees to their data usage (opt-in) and Y denies it (opt-out). We create the Purpose Filter with X's identifier (a row id or tuple id) in the positive set and Y's identifier in the negative set. Those ids are inserted in the Purpose Filter according to the algorithm and hashed within the appropriate layers of the filter. This models the consent for Purpose P for tuples X and Y.

As a second example, let us have the same setup, but now with purposes in attributes A and B, P_A and P_B. X and Y will have cell-level consent, acknowledging the usage of their data in both attributes. Now, let's assume that X consents to both attributes, but Y only consents to attribute A and denies attribute B. The filter for P_A will have X and Y ids in the positive set, but the filter for P_B will have X in the positive set and Y in the negative set. This will change query results, as we show below.

To embed the Purpose Filter in the DBMS, the execution operators must be aware and include probing the filter with tuple IDs in query processing. Upon query resolution, the scan operator should evaluate whether there is a filter associated with the table or attributes involved in the query so it can know which tuples are to be retrieved upon execution.

From our example, let us take two queries posed against this system, each pertaining

to the given purpose:

```
Q1 = SELECT * FROM TABLE T

Q2 = SELECT A, B FROM TABLE T
```

To answer $Q1$, we have the first scenario, the same one presented in our running example. The scan operator retrieves the filter from the table and checks the unique IDs against it to decide whether a tuple should be returned from the query.

To answer $Q2$, we have the second scenario. The scan operator must perform two checks against two filters, one for attribute A and one for attribute B. If a register is partially retrieved (i.e., user 5 consented to attribute A but denied attribute B, or if B returned as a false negative), the query processor may add a $NULL$ value to the respective column, thus complying with the definitions established.

Although both queries used in this discussion are simple queries, the consent enforcement happens at the lowest level of the query execution tree, therefore, the input that other operators receive from the scan operator is already compliant, and will remain compliant, since unauthorized data has been already filtered out and no new data is added from tables with personal data.

To avoid redundant checks on the filter, a query or transaction may hold the intermediate result from the scan operator in memory and reuse it, reducing the overhead on more complex queries, such as nested queries or long transactions.

4.9 Summary

In this chapter, we discussed our data representation structure, the Purpose Filter, a filter-based data structure that allows us to represent purpose metadata and query personal data without violating purpose properties and definitions. We presented the construction of the Purpose Filter, which is built by layering filters, alternating between positive and negative layers. The probing function ensures that only allowed tuples are returned, achieving a zero false positive rate. We also discussed the maintenance of the filter, including how to handle updates and deletions, and the costs associated with building and querying the filter. Finally, we modeled the information loss associated with purpose-based access control and provided an overview

of how Purpose Filter integrates with DBMS query processing to enforce consent and purpose compliance.

5 DIFFERENTIALLY PRIVATE COUNTING BLOOM FILTERS

As mentioned in Chapter 3, only Bloom Filters were covered under differential privacy studies. We further this coverage by developing DPCBF, a differentially private counting Bloom filter.

Both data structures work similarly; instead of an array of bits, we have an array of non-negative integers (counters) and a group of hash functions. Counting Bloom Filters allow for delete operations, and may function to answer aggregate queries, for example, to know how many elements were inserted in a CBF, one must simply add up all the counters and divide by the number of hash functions used in the filter.

Previous works (ALAGGAN *et al.*, 2012; ERLINGSSON *et al.*, 2014; KE *et al.*, 2025) used randomized response as the differential privacy mechanism. However, this becomes inappropriate when we have counters instead of bits. For our structure, we also have a restriction that our added noise must be integer-valued, to keep the filter correct and consistent.

We first construct the filter using the elements to be added, then add a two-sided geometric noise using the geometric mechanism. By this data perturbation, we ensure differential privacy, as will be proved in the next subsection. Algorithm 3 shows how it is constructed, and Algorithm 4 shows how it is queried.

Algorithm 3: DP Counting Bloom Filter constructor function

Data: Elements $S \subset U$, hash functions H, privacy budget ε, array of integers cbf of size m

Result: DP-CBF filter

```
1 cbf ← {0}^m foreach s ∈ S do
2 │  foreach h ∈ H do
3 │  │  cbf[h(s)] ← cbf[h(s)] + 1
4 │  end
5 end
6 for i ← 0 to m do
7 │  cbf[i] ← cbf[i]+ geometric noise with budget ε/k
8 end
9 return cbf;
```

Algorithm 4: DP Counting Bloom Filter query function

```
   Data: Element x ∈ U, hash functions H, array of integers cbf of size m
   Result: boolean True if x ∈ S, False otherwise
1  foreach h ∈ H do
2  |  if cbf[h(x)] ≤ 0 then
3  |  |  return False
4  |  end
5  end
6  return True;
```

5.1 Privacy Guarantees

Algorithm 3 presents the application of the geometric mechanism in the construction of a CBF, providing protection of privacy for the elements used in this process. In line 7, after the insertion of all elements in the CBF, we alter the count values of all cells in the array independently by adding a geometric noise using a budget $\varepsilon' = \frac{\varepsilon}{k}$ where k is the number of hash functions used. Theorems 5.1 and 5.2 show how this noise addition achieves differential privacy.

Theorem 5.1. The composability of the mechanism is serial concerning the number of hash functions and parallel for the number of cells in the filter.

Proof. Take a counting Bloom filter X and an element from the input set, $\{i\}$,. Take H as the set of hash functions and $k = |H|$ as the number of hash functions. We want to show that the minimal ε is $\frac{\varepsilon}{k}$.

If we partition *X(i)* in two, being the first partition $X(i)^T$ correspondent to all the positions where $H(i)$ maps to, and $X(i)^{-T}$ as the second partition having all the positions not mapped by $H(i)$, we have, since T and $-T$ are complementary:

$$Pr(M(X(i)) = r) = Pr(M(X(i)^T) = r^T) * Pr(M(X(i)^{-T}) = r^{-T})$$

Exploring X in terms of i, we know that i^T maps to k positions, while i^{-T} contains the rest of all non-k positions. Generalizing for all i, we have that X^{-T} is not affected by a given i. Since it is independent, the relevant noise is added on $Pr(M(X^T) = r^T)$, and because an i only affects k positions, the sequential composition happens only on k the budget should be split in $\frac{\varepsilon}{k}$. As every i is independent, the composition is parallel on i and on each cell of the filter.

□

Theorem 5.2. By adding a geometric noise with budget ε and sensitivity k, we achieve ε-differential privacy.

This theorem encompasses two different proofs:

1. Proof that each cell of the array is private
2. Proof that the query algorithm is private

And we have the following premises:

- Take the neighboring datasets *X* and *Y*, differing on one element g, such that $X = Y \bigcup \{i\}$.
- Let cbf be the counting Bloom filter generated from set *X* and cbf' be the counting Bloom filter generated from set *Y*.
- Let $\varepsilon' = \frac{\varepsilon}{|H|} > 0$
- Let $dpcbf$ be the noisy counting Bloom filter generated from set *X* and $dpcbf'$ be the noisy counting Bloom filter generated from set *Y*.
- Let $dpcbf[i] \in \mathbb{Z}$ be the value of the noisy answer on position *i*.

Let us begin with proof 1:

Proof. We will show that $\forall i \in [1...m]$, $dpcbf[i]$ is ε'-DP.

$\forall i \in [1...m]$, $cbf[i]$ is the i-th value stored in the CBF. We must have $\frac{Pr[dpcbf[i]=r]}{Pr[dpcbf'[i]=r]} \leq e^{\varepsilon'}$. r can be seen as the true value $h(i)$, plus a geometric noise Z_g.

$$\frac{Pr[dpcbf[i] = r]}{Pr[dpcbf'[i] = r]} \leq e^{\varepsilon'}$$

$$dpcbf[i] = cbf[i] + Z_g$$

$$dpcbf'[i] = cbf'[i] + Z_g$$

$$\frac{Pr[cbf[i] + Z_g = r]}{Pr[cbf'[i] + Z_g = r]} \leq e^{\varepsilon'}$$

There are two cases: $dpcbf[i] = dpcbf'[i]$ and $dpcbf[i] \neq dpcbf'[i]$.

For the first case, we have the same values for both $cbf[i]$ and $cbf'[i]$, since Z_g is the same, therefore:

$$\frac{Pr[dpcbf[i] = cbf[i] + Z_g]}{Pr[dpcbf'[i] = cbf'[i] + Z_g]} \leq e^{\varepsilon'}$$

$$1 \leq e^{\varepsilon'} \tag{5.1}$$

For the second case, $cbf[i]$ and $cbf'[i]$ differ by 1, because of g hitting the *i*-th index in cbf and not in cbf', since it is absent from cbf'. Take $cbf[i] + Z_g = k$. Take $p = \exp(-\varepsilon')$

$$\frac{Pr[dpcbf[i] = cbf[i] + Z_g]}{Pr[dpcbf'[i] = cbf'[i] + Z_g]} \leq e^{\varepsilon'}$$
$$\frac{Pr[Mg(dpcbf[i]) = k - cbf[i]]}{Pr[Mg(dpcbf'[i]) = k - cbf'[i]]} \leq e^{\varepsilon'}$$
$$\frac{\frac{1-p}{1+p} * p^{|k-cbf[i]|}}{\frac{1-p}{1+p} * p^{|k-cbf'[i]|}} \leq e^{\varepsilon'}$$
$$\frac{p^{|k-cbf[i]|}}{p^{|k-cbf'[i]|}} \leq e^{\varepsilon'}$$
$$\frac{p^{|k-cbf[i]|}}{p^{|k-(cbf[i]-1)|}} \leq e^{\varepsilon'}$$
$$\frac{p^{|k-cbf[i]|}}{p^{|k+1-cbf[i])|}} \leq e^{\varepsilon'}$$

Since we have an absolute value, to factorize, we must consider both signs.

$$\frac{p^{|k-cbf[i]|}}{p^1 * p^{|k-cbf[i])|}}$$
$$p^{-1} = \exp(-1(-\varepsilon')) = e^{\varepsilon'} \quad (5.2)$$
$$\frac{p^{|k-cbf[i]|}}{p^{-1} * p^{|k-cbf[i])|}}$$
$$p^1 = e^{-\varepsilon'} \quad (5.3)$$

Therefore, because $e^{-\varepsilon'} \leq p \leq e^{\varepsilon'}$ for all values of i, $dpcbf[i]$ is ε'-DP.

□

Now for proof 2:

Proof. The query algorithm outputs either $true$ or $false$, based on the value of each hit cell. Take A, B as two sets such that $A \bigcup B$ is equal to all possible values of i. A will hold all values of i where $cbf[i] = cbf'[i]$ and B the remaining values such that $cbf[i] \neq cbf'[i]$. One point to consider in B is that, like the previous proof, $cbf[i]$ and $cbf'[i]$ differ by 1.

Since we aim for the same answer from either dataset, differential privacy states that the ratio between the probabilities of both datasets outputting the same result must be bounded by e^{ε}. For elements in A:

$$\frac{Pr[cbf[i] + Z_g = v]}{Pr[cbf'[i] + Z_g = v]} = \frac{Pr[cbf[i] + Z_g > 0] + Pr[cbf[i] + Z_g \leq 0]}{Pr[cbf'[i] + Z_g > 0] + Pr[cbf'[i] + Z_g \leq 0]}$$

Because the probabilities are symmetrical, we pick one side to simplify to

$$\frac{Pr[cbf[i] + Z_g = v]}{Pr[cbf'[i] + Z_g = v]} = \frac{Pr[cbf[i] + Z_g > 0]}{Pr[cbf'[i] + Z_g > 0]} = 1 \tag{5.4}$$

As we have $cbf[i] = cbf'[i]$.

Now for *B* we have:

$$\frac{Pr[cbf[i] + Z_g = v]}{Pr[cbf'[i] + Z_g = v]} = \frac{Pr[cbf[i] + Z_g > 0] + Pr[cbf[i] + Z_g \leq 0]}{Pr[cbf'[i] + Z_g > 0] + Pr[cbf'[i] + Z_g \leq 0]}$$

Like before, we simplify to:

$$\frac{Pr[cbf[i] + Z_g \leq 0]}{Pr[cbf'[i] + Z_g \leq 0]}$$

Where $cbf[i] = cbf'[i] \pm 1$

Taking $cbf[i] = cbf'[i] + 1$, from the geometric distribution we have, since our real values are non-negative:

$$Pr[cbf[i] + Z_g \leq 0] = \frac{p^{cbf[i]}}{1 - p}$$

Substituting:

$$\frac{Pr[cbf[i] + Z_g \leq 0]}{Pr[cbf'[i] + Z_g \leq 0]} =$$

$$\frac{\frac{p^{cbf[i]}}{1+p}}{\frac{p^{cbf'[i]}}{1+p}} =$$

$$\frac{\frac{p^{cbf[i]}}{1+p}}{\frac{p^{cbf[i]+1}}{1+p}} =$$

$$\frac{p^{cbf[i]}}{p^{cbf[i]+1}} = \frac{1}{p} \tag{5.5}$$

Now for $cbf[i] = cbf'[i] - 1$

$$\frac{\frac{p^{cbf[i]}}{1+p}}{\frac{p^{cbf'[i]}}{1+p}} =$$
$$\frac{\frac{p^{cbf[i]}}{1+p}}{\frac{p^{cbf[i]-1}}{1+p}} =$$
$$\frac{p^{cbf[i]}}{p^{cbf[i]-1}} = p \tag{5.6}$$

And p is $e^{\varepsilon'}$.

Now for $A \bigcup B$

$$\prod_{i \in A} \frac{Pr[dpcbf[i] = v]}{Pr[dpcbf'[i] = v]} \prod_{i \in B} \frac{Pr[dpcbf[i] = v]}{Pr[dpcbf'[i] = v]}$$
$$\prod_{i \in A} \frac{Pr[dpcbf[i] = v]}{Pr[dpcbf'[i] = v]} = 1$$
$$\prod_{i \in B} \frac{Pr[dpcbf[i] = v]}{Pr[dpcbf'[i] = v]} = e^{|B| * \varepsilon'}$$
$$\ln e^{|B| * \varepsilon'} = |B| * \varepsilon'$$
$$|B| * \frac{\varepsilon}{k} \tag{5.7}$$

Since B represents the number of positions changed in a neighboring dataset, $|B|$ contains exactly k elements, and by substituting in (12), we get that the ln of the probability equals ε, thus finalizing our proof of ε-DP for the query algorithm.

□

5.2 Utility Evaluation

To analyze utility, we will compare our answers to a non-perturbed counting Bloom filter. We want to calculate how probable it is that both filters yield the same answer. To do so, take $Q = \{j \in [m] : h_i(q) = j, \forall i \in H\}$. Q is the set of indices reached for every query q. Each j corresponds to an index, and each i to a given hash function in the filter. Also, let $Eval(Q, f) = \{true, false\}$ be a function that answers true if all $f[j] > 0$ and false otherwise. Therefore, we can model our probability as:

$$Pr[Eval(Q, dpcbf) = Eval(Q, cbf)]$$

Using conditional probability rules, we have:

$$Pr[Eval(Q, dpcbf) = Eval(Q, cbf)] =$$
$$Pr[Eval(Q, dpcbf) = true|Eval(Q, cbf) = true] * Pr[Eval(Q, cbf) = true] +$$
$$Pr[Eval(Q, dpcbf) = false|Eval(Q, cbf) = false] * Pr[Eval(Q, cbf) = false]$$

Let us observe that to have the same result, we must have $((dpcbf[j] > 0) \wedge (cbf[j] > 0)) \vee ((dpcbf[j] <= 0) \wedge (cbf[j] = 0))$. The query function in Algorithm 4 covers the case $(dpcbf[j] <= 0)$ and by definition, $cbf[j]$ is non-negative; therefore, for the query result, both return $false$. Remember that $dpcbf[j] = cbf[j] + Z_g$. Let us calculate each probability:

For $Pr[Eval(Q, dpcbf) = true|Eval(Q, cbf) = true]$ we have:

$$\begin{aligned} Pr[Eval(Q, dpcbf) = true|Eval(Q, cbf) = true] &= \prod_{i=1}^{k} Pr[dpcbf[Q[i]] > 0 \wedge cbf[Q[i]] > 0] \\ &= \prod_{i=1}^{k} Pr[cbf[Q[i]] > -Z_g] \\ &= \prod_{i=1}^{k} \frac{1 + e^{\varepsilon'} - e^{\varepsilon' * cbf[Q[i]]}}{1 + e^{\varepsilon'}} \end{aligned} \tag{5.8}$$

For $Pr[Eval(Q, dpcbf) = false|Eval(Q, cbf) = false]$, we must provide new definitions. Let $Zr = \{j \in Q : f[j] = 0\}$ be the set of indexes where $f[j] = 0$. Let $O = \{j \in Q : f[j] > 0\}$ be the set of indexes where $f[j] > 0$. Observe that $Zr \bigcup O = Q$. By probability rules, $Pr[Eval(Q, dpcbf) = false|Eval(Q, cbf) = false] = 1 - Pr[Eval(Q, dpcbf) = true|Eval(Q, cbf) = false]$.

$Pr[Eval(Q, dpcbf) = true|Eval(Q, cbf) = false]$ only holds if:

1. All elements in Zr receive a positive geometric noise, i.e., $Z_g > 0$.
2. No elements in O receive a negative noise greater than their value, i.e., $value + Z_g > 0$

From these conditions, we have:

$$
\begin{aligned}
Pr[Eval(Q, dpcbf) = true | Eval(Q, cbf) = false] &= \prod_{i=1}^{|Zr|} Pr[Z_g > 0] * \prod_{i=1}^{|O|} Pr[cbf[O[i]] > -Z_g] \\
&= \prod_{i=1}^{|Zr|} Pr[Z_g > 0] * \prod_{i=1}^{|O|} Pr[cbf[O[i]] > -Z_g] \\
&= \prod_{i=1}^{|Zr|} \frac{e^{\varepsilon'}}{1 + e^{\varepsilon'}} * \prod_{i=1}^{|O|} \frac{1 + e^{\varepsilon'} - e^{\varepsilon' * cbf[O[i]]}}{1 + e^{\varepsilon'}}
\end{aligned} \tag{5.9}
$$

The probabilities for $Eval(Q, cbf) = true$ and $Eval(Q, cbf) = false$ come straight-forward from the Counting Bloom filter, because the algorithm is the same, so:

$$
\begin{aligned}
Pr[Eval(Q, cbf) = true] &= Pr[cbf[j] > 0 \forall j \in Q] \\
&= 1 - e^{\frac{kn}{m}}
\end{aligned} \tag{5.10}
$$

$$
\begin{aligned}
Pr[Eval(Q, cbf) = false] &= 1 - Pr[cbf[j] > 0 \forall j \in Q] \\
&= e^{\frac{kn}{m}}
\end{aligned} \tag{5.11}
$$

Substituting 5.8, 5.9, 5.10, 5.11 into our final equation:

$$
\begin{aligned}
Pr[Eval(Q, dpcbf) = Eval(Q, cbf)] = &\left(\prod_{i=1}^{k} \frac{1 + e^{\varepsilon'} - e^{\varepsilon' * cbf[Q[i]]}}{1 + e^{\varepsilon'}}\right) * (1 - e^{\frac{kn}{m}}) + \\
&(1 - (\frac{e^{\varepsilon'}}{1 + e^{\varepsilon'}})^{|Zr|} * (\prod_{i=1}^{|O|} \frac{1 + e^{\varepsilon'} - e^{\varepsilon' * cbf[Q[i]]}}{1 + e^{\varepsilon'}})) * (e^{\frac{kn}{m}})
\end{aligned} \tag{5.12}
$$

5.3 Summary

In this chapter, we presented the DPCBF, a differentially private counting Bloom filter. We showed how to construct it using a geometric mechanism, and proved that it is ε-DP. We also presented a utility evaluation, showing how probable it is that the DPCBF yields the same answer as a non-perturbed counting Bloom filter. As Purpose Filter is compatible with Counting Bloom Filter for its layers, we designed DPCBF to be an alternative when applying differential privacy for filters and be able to use it in conjunction with Purpose Filter. Our experimental

evaluation in Chapter 7 will show that DPCBF performs better than the DP-Bloom Filter, which is the only other differentially private filter available in the literature, improving the overall usefulness for a differentially private Purpose Filter, which we discuss in the next chapter.

6 DIFFERENTIALLY PRIVATE PURPOSE FILTER

In this chapter, we discuss how to add differential privacy to Purpose Filter, by adding noise to a layer and what it entails.

6.1 Filter Construction

As discussed in Chapter 1, our goal is to protect users' consent metadata, avoiding re-identification on who provided what consent value. In Chapter 4, we discussed a novel way of storing consent metadata, Purpose Filter, and in Chapter 5, we discussed how to turn a filter differentially private, specifically Counting Bloom Filters, which can be used as layers of Purpose Filter.

With these two concepts, protecting consent metadata stored in Purpose Filter should be straightforward. However, as highlighted in chapter 5, adding noise to a filter allows for both types of error, false positives and false negatives. Remember that our problem does not allow for false positives, because if it did, it would result in non-consented data being retrieved by the DBMS.

If DP is applied directly to our data structure, what would happen is that positive and negative layers would yield both types of error. To clarify, let us look at the following scenario:

- A Purpose Filter constructed with one pair of layers
- A set of positive and negative consents

Normally, as shown in chapter 4, we have a 0 FPR for this set, and all is good. However, if we simply add DP to both layers, following the process of constructing the layer, then adding noise, what would happen is:

- For the positive layer, it could yield a false positive (a negative item being accepted) or a false negative (a positive item being rejected). In both cases, there is no violation of consent.
  - If a negative item is accepted by a positive layer, we have the negative layer as a fallback, so far.
  - If a positive item is rejected by a positive layer, we may increase our FNR, but consent is not violated.
- For the negative layer, the false positive means a positive item being accepted, and the false negative a negative item being rejected. This causes a violation of consent.

  - If a positive item is accepted by a negative layer, it is seen as negative, impacting only the FNR, but consent is not violated.
  - If a negative item is rejected by a negative layer, it is seen as positive. This violates consent.

From this scenario, we have the following takeaways:

- If we add noise to every layer, consent may be violated.
- Noise on the positive layer could increase the FNR, but does not violate consent.

Note that building the filter and adding noise to the positive layer(s) does not guarantee privacy for the whole filter, in the sense that the negative layers are still unprotected. To solve this issue, we change the construction algorithm:

Algorithm 5 differs from Algorithm 1 only in lines 11 to 15. As we will show, these lines ensure DP by perturbing the first layer of Purpose Filter, after the positive set is inserted. Depending on the type of filter, the DP mechanism may change. If we have a Bloom Filter for layer, we can use the Randomized Response; if there is a counting Bloom Filter, we may use the DPCBF presented in Chapter 5.

Figure 17 exemplifies the process. As we follow the algorithm, the first layer is created with the positive elements inserted, as shown on the top left. Then, moving to the top right figure, this layer is perturbed, protecting the positive elements. When the negative set is probed, in the bottom left figure, the answer from the positive layer is differentially private, making the survivor set DP. Bottom right shows a query example for the positive set, which now returns negative for 1 (accepted by all layers) and 2 (rejected by the positive noisy layer), and positive for 3 and 4 (both rejected by the negative layer).

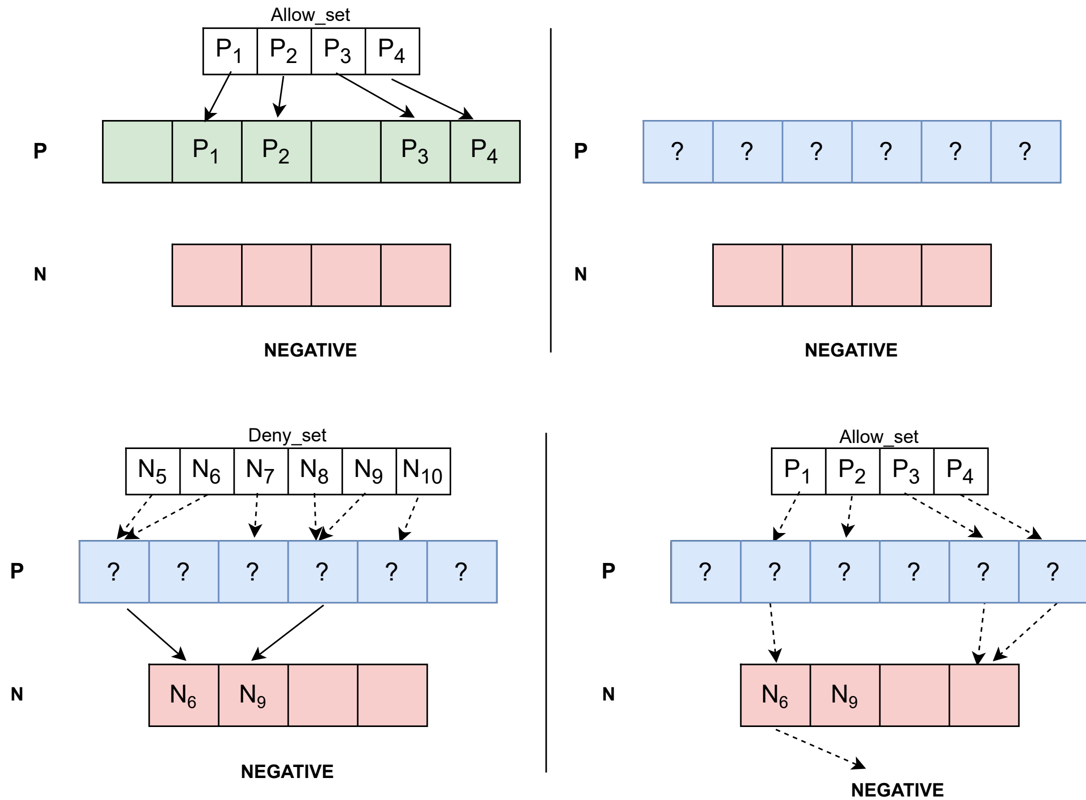
Different from the non-perturbed Purpose Filter, shown in figure 15, our survivor set changes because the first layer now may accept other false positives. The FNR will also change, given that the first layer now yields false negatives.

Also, observe that the perturbation is made only on the first layer. This is enough to guarantee DP. We could perturb every positive layer, but would only increase noise and decrease utility, on an already DP data structure.

Algorithm 5: DPPF Constructor function

```
    Data: Positive set P, Negative set N, False Positive Rate FPR, # of hashes k, privacy
          budget ε
    Result: Differentially Private Purpose Filter
 1  filter = ;
 2  finished = false;
 3  filter.layers = {};
 4  while finished == false do
 5      R_set = {} // Elements that survive probing
 6      if filter.layer_counter % 2 == 0 then
 7          build layer with appropriate size ;
 8          for i=0 to P.length do
 9              layer.insert(P[i]);
10          end
11          if is the first layer then
12              for each position in the filter do
13                  add appropriate noise with budget ε;
14              end
15          end
16          for j=0 to N.length do
17              if layer.probe(N[j]) == true then
18                  R_set.add(N[j])
19              end
20          end
21          N = R_set // Update negative elements to only survivors
22          filter.layers.add(layer);
23      end
24      else
25          build layer with appropriate size;
26          for i=0 to N.length do
27              layer.insert(N[i]);
28          end
29          for j=0 to P.length do
30              if layer.probe(P[j]) == true then
31                  R_set.add(P[j])
32              end
33          end
34          P = R_set // Update positive elements to only survivors
35          filter.layers.add(layer);
36          finished = FNR_is_ok();
37      end
38  end
39  return filter;
```

Figure 17 – DP-Purpose Filter construction (reads left-to-right)



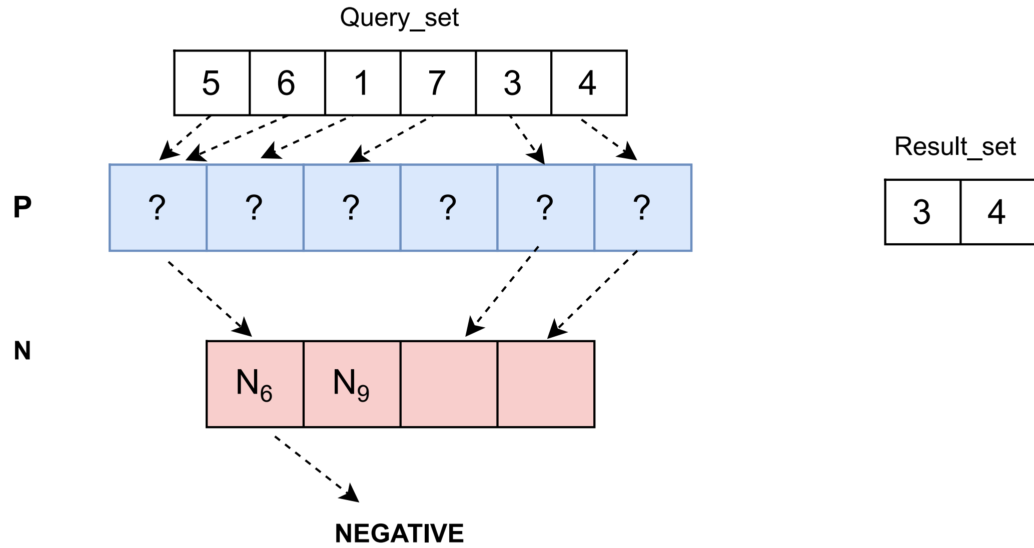
Source: elaborated by the author.

6.2 Filter Probing

The probing algorithm is the same as Algorithm 2; the result is what may change. What acceptance and rejection mean for each layer does not change, and the first layer is still considered positive. To recall: if the first layer accepts an element, it is deemed positive and moves to be tested by the second layer; if it rejects, the element is deemed negative. Figure 18 shows the filter probing process:

The probing begins in the first layer, now differentially private. From our query set, elements 5 and 6 collide in the first position, which accepted them, so they fall through the next layer as they were added as negative and correctly return negative. Element 1 is rejected by the positive layer and returns negative, since it is positive, this is a false negative. Element 7 is correctly rejected by the positive layer and returns negative. Elements 3 and 4 are accepted by the positive layer but rejected by the negative layer, returning positive, as they are. In the end, the returned tuples are 3 and 4, which are true positives. 1 is ignored as a false negative, and no

Figure 18 – DP-Purpose Filter probe



Source: elaborated by the author.

true negatives are returned.

In comparison with the non-perturbed example, in Figure 16, we note that the overall query result changed. Nevertheless, the query result is still correct, as we are returning only true positives.

As mentioned before, our first layer now yields false positives and negatives, which will increase the overall FNR of the filter, because some true positives may be rejected by the first layer as false negatives when probing.

Compared with Figure 16, Figure 18 shows that the first layer perturbation changes the query output. Before, we returned {1,4} as our result, and 2 was considered a false negative by the filter. Now, because of the perturbation, 1, which was a true positive, is now rejected by the first layer of the filter. 3 and 4 are accepted by the first layer and rejected by the second, returning as positives, which they are, true positives. 5 and 6 are accepted by the first layer, but the second captures their true behavior, and they reach the end of the filter, returning negative, which they are.

This example shows that our information loss is not the same as the non-perturbed filter. In the next section, we explore the changes and model the new information loss equation.

6.3 Information Loss

Our new information loss needs to account for the false negatives now yielded by the first layer, alongside the acceptance of positives by the negative layer. For the first layer, we can calculate the probability of false negatives from equation 5.8. We can assume a false negative if the original answer is positive and the perturbed answer is negative. From conditional probability rules:

$$Pr[Eval(Q, dpcbf) = false | Eval(Q, cbf) = true] =$$
$$1 - Pr[Eval(Q, dpcbf) = true | Eval(Q, cbf) = true]$$

Then, we can get the total FNR for the DP filter as:

$$TotalDPFNR = 1 - Pr[Eval(Q, dpcbf) = true | Eval(Q, cbf) = true] + TotalFNR \quad (6.1)$$

Where $Pr[Eval(Q, dpcbf) = true | Eval(Q, cbf) = true]$ comes from 5.8 and $TotalFNR$ comes from 4.1.

6.4 Privacy Guarantees

Our privacy guarantees rely strongly on the post-processing theorem. Let us analyze what the guarantees are for the positive and negative sets:

For the positive set, the privacy guarantee is straightforward, coming from the privacy guarantee of the mechanism applied to the data structure, because we construct the first layer with the positive set and then add noise. It follows that every item in this data structure is protected because it is now a DP structure.

For the negative set, we use the post-processing theorem. After perturbing the positive layer, we take the negative set and probe it, storing the survivors. As the theorem states, if we have an ε-DP algorithm, applying a function to this algorithm also satisfies ε-DP. Since we have an ε-DP filter, the result of the probe function will also be ε-DP. Therefore, the negative survivor set is ε-DP. Every other step in the construction of the filter will start from this negative survivor set, and the positive set will be tested against a negative layer constructed from this set.

In summary, everything that comes after the construction of the first layer is ε-DP because the first layer is used in the process.

We provide an experimental evaluation on the impact of FNR due to added noise, comparing both types of filters for which we can ensure DP: Bloom Filters, from other works, and Counting Bloom Filters, from this work.

7 EXPERIMENTAL EVALUATION

In this section, we empirically evaluate Purpose Filter, DPCBF, and DPPF. Experiments were run in a Lenovo ThinkPad machine with an Intel Core I7 10510U 1.8 GHz, with 16GB of RAM, using JVM version 21. Different sets of experiments were performed for each approach, as follows:

- Purpose Filter: We evaluate the storage space used by the filter, the false negative rate, and the number of layers and their sizes. We also compare space allocation with other approaches, such as Sieve (PAPPACHAN *et al.*, 2020) and Hippocratic Databases (AGRAWAL *et al.*, 2002).
- Differentially Private Counting Bloom Filters (DPCBF): We show what the perturbed filter looks like in comparison to an unperturbed one. We evaluate how the added geometric noise and configurations impact the DPCBF with respect to false positives and false negatives, performing a direct comparison with DPBloomFilter (KE *et al.*, 2025). We also evaluate the impact of adding DP in the reconstruction attack and the usage of DPCBF in a polling application.
- Differentially Private Purpose Filter (DPPF): We evaluate the error variation with the added noise, using two different configurations for the DP layer. DP-Bloom Filters and DP-Counting Bloom Filters. We also evaluate an ML application training using a standard consented dataset, the same consents stored in Purpose Filter and DPPF, comparing the prediction error of the models.

7.1 Purpose Filter

A first version of Purpose Filter was implemented alongside Purpose Scan (PRACIANO *et al.*, 2022). The current version was implemented using Java 21, where the following experiments were performed.

7.1.1 Evaluation Setup

To represent the personal data tuple IDs in the database, we generated 10 million random unique integers. This set was then partitioned into our positive and negative sets according to the experiment proportion. To remind, the positive tuple IDs are the ones that consent to their data being used for a specific purpose, while the negative tuple IDs are the ones that do not

consent.

We used Bloom Filters (BLOOM, 1970) as the filter layers of the Purpose Filter. We used Google Guava's fixed MurmurHash3 implementation with 32 bits as our hash function. To calculate the Nth hash positions, we used the double hashing technique from Mitzenmacher (MITZENMACHER, 2014). Every experiment was run 5 times, and the presented results are the average of these runs.

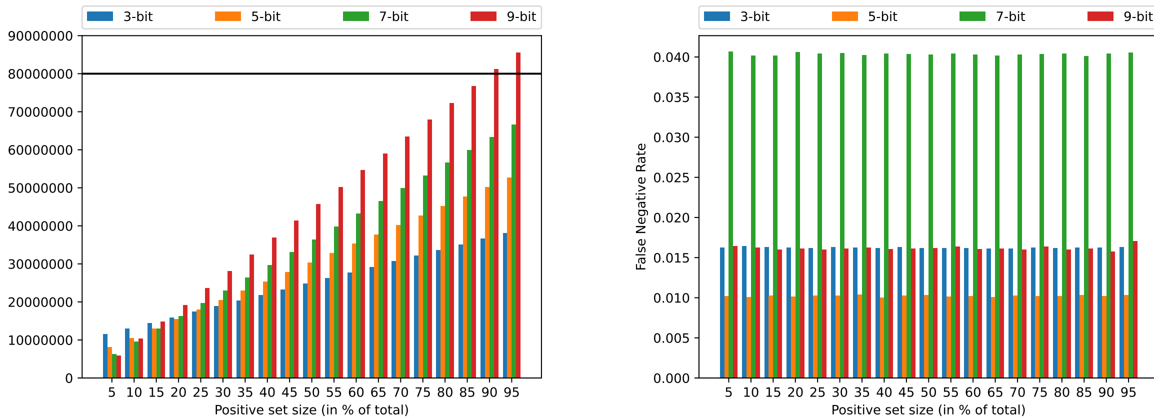
We compare how much space is used by the data structure in comparison with an approach of storing each purpose alongside the tuple as a 1-byte attribute, simulating a foreign key binding that tuple to a purpose table. We partition our dataset into x% positives and (100-x)% negatives. We vary the proportion in increments of 5%, beginning at 5% and going until 95%. We perform 2 experiments, one with a set amount of bits and one with a target FPR for the first layer, calculating the optimal number of bits and storage size. We also compare the storage overhead with Sieve (PAPPACHAN *et al.*, 2020) and Hippocratic Databases, using the implementation from (KONSTANTINIDIS *et al.*, 2021). Each experiment in this section is executed 5 times, with a new random set generated each time. The average of the runs is presented in the result.

7.1.2 Fixed size

In this subsection, the experiments discuss the configuration, where we set the number of bits used to represent each element in four configurations: 3-bit, 5-bit, 7-bit, and 9-bit. **Storage Space.** In this experiment, we observe how the data structure grows according to the proportion of positive and negative elements. Figure 19a presents how the space consumption varies in this window. The x-axis represents the proportion of the positive set, seen as a sliding window (5% positives, 95% negatives; 10% positives, 90% negatives, and so on). The y-axis represents the amount of space consumed in bits compared to the baseline approach. The legend indicates the configuration, from left to right corresponding to the stated configurations above. The bar above indicates the storage used by our baseline, which is constant (8-bits times 10m tuples).

We observe that the amount of storage space used increases with the positive set proportion, i.e., when there are more positive elements, the filter size increases. This happens because the first layer has all positives inserted into it, and while the filter sees all negative elements during construction, only the negative elements accepted by the first layer during construction are added to the negative layer.

Because of the small proportion of positive items in the beginning, the first layer in



(a) Variation of space used. (b) Variation of FNR.

Figure 19 – Experiments for fixed size

lower-bit configurations yields more false positives. Therefore, the negative survivor set becomes very large. With more bits, this phenomenon ceases to exist, which is why, as the proportion becomes more balanced, the expected behavior (more bits = bigger filter size) becomes the norm.

Another interesting observation is that even with 9 bits per element, which would be more than the baseline, the total filter size only overcomes the baseline when there are more than 90% positives. This result comes from our filter construction, which leans more heavily on the positive set size, instead of the baseline, which would be a constant value.

Knowing that the amount of bits used to represent elements impacts the False Negative Rate, we proceed to the next experiment.

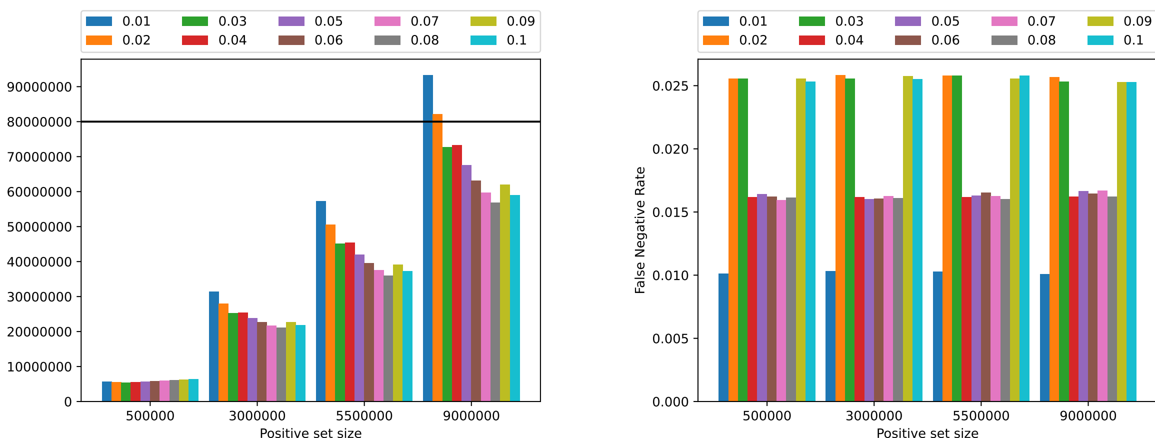
False Negative Rate. In this experiment, we observe how the False Negative Rate (FNR) varies when the proportion of positive and negative elements changes. Figure 19b presents how this FNR varies over. The x-axis represents the size of the positive set. The y-axis represents the percentage of false negatives returned by the filter. The legend indicates the configuration, from left to right, equal to the stated configurations above. The build threshold was set as a 5% false negative rate for the filter.

The FNR remains very close for different positive set sizes and the same configuration, showing resilience from the filter with varying positive set sizes. The variation shown, especially on the 7-bit configuration, comes from the number of layers. While the 3 and 5-bit had 6 layers, therefore, they needed three iterations of the algorithm to finish, and the 7-bit finished with 2 layers, marking the turning point of the decision process. Upon observing this result, we executed with a threshold of 3%, and the FNR falls to around 0.1%, but it builds 4 layers instead of 2.

7.1.3 *Fixed* FPR_0

In this experiment, we fix the FPR of the first layer, from 1% to 10%, and observe how the full structure behaves regarding storage space and FNR. Given the amount of data to be presented, we select four positive set proportions, 5%, 30%, 55%, 90%, to present our data.

Storage Space. In this experiment, we observe how the data structure grows according to the proportion of positive and negative elements. Figure 20a presents how space consumption varies over the selected values. The x-axis represents the size of the positive set. The y-axis represents the amount of space consumed in bits when compared to the baseline approach. The legend indicates the configuration, from left to right, equal to the stated configurations above. The bar above indicates the storage used by our baseline, which is constant (8 bits times 10m tuples).



(a) Variation of space used.

(b) Variation of FNR.

Figure 20 – Experiments for fixed FPR_0

We observe that the amount of storage space used decreases with increasing FPR_0. This happens because to achieve a low FPR on the first layer, we need to allocate more bits. This behavior is more apparent when the positive set size is large enough.

We also observe a slight bump up from 8% to 9%. That is also due to the threshold, where 8% has 6 layers in the filter and 9% goes to 8 layers, adding a little more size.

The storage size comparison to the baseline was already discussed; therefore, we skip it to avoid repetition.

False Negative Rate. In this experiment, we observe how the False Negative Rate (FNR) varies when the proportion of positive and negative elements changes. Figure 20b presents how this FNR varies over the proportion changes. The x-axis represents the size of the positive

set. The y-axis represents the percentage of false negatives returned by the filter. The legend indicates which FPR was fixed for the first layer. Once again, the build threshold was set to 5%.

By fixing an FPR to the first layer, we add a step of figuring out the optimal size of the Bloom Filter layer. We also observed that although the subsequent layers are not bound and yield a rather large FPR, the full filter FNR remains bounded. This is also reflected in space-saving for filter storage.

Going back to Figure 20b, we observe that we do have a rise in the FNR for the 0.02 and 0.03 FPR configurations. Upon closer observation, we see that this is also when the construction threshold is near; with the 0.04 FPR onwards, the construction process adds 2 more layers, further reducing the FPR. What we also observe is that although the space required to store the filter decreases when we advance on the FPR, the FNR remains mostly constant. This behavior happens because of the calculation of the optimal size. Since we use a *log* base 2, the number of bits remains mostly unchanged from 0.04 to 0.08, highlighting that upper bounds may give the best cost-benefit relationship with regard to FNR and storage size.

7.1.4 Experiments for number of layers and layer sizes

In this experiment, we dissect the constructed filters under selected configurations to analyze the number of layers and their sizes to verify what happens with some configurations that stand out from our experiments. Tables 2 and 3 present these results.

Table 2 – Number of layers and pair sizes for different fixed sizes.

Config	# layers	Pair 1 size	Pair 2 size	Pair 3 size	Pair 4 size	Total Size
3-bit	6	19,910,272	5,037,184	1,277,312	-	26,224,768
5-bit	4	29,770,496	3,008,512	-	-	32,779,008
7-bit	2	39,773,888	-	-	-	39,773,888
7-bit*	4	39,772,416	1,614,720	-	-	41,387,136

Table 3 – Number of layers and pair sizes for different fixed FRRs.

Config	# layers	Pair 1 size	Pair 2 size	Pair 3 size	Pair 4 size	Total Size
0.02 FRR	4	46,998,400	3,566,272	-	-	50,564,672
0.04 FRR	6	39,962,432	4,308,096	1,092,352	-	45,362,880
0.08 FRR	6	30,349,888	4,435,840	1,121,216	-	35,906,944
0.1 FRR	8	29,838,016	4,769,216	1,911,040	768,512	37,286,784

We selected four configurations from the fixed size and four from the fixed FRR. To maintain a standard of comparison, all data were collected using the positive set size 55%. From

the first table, we already see that more layers do not necessarily mean that the filter will be larger. As we have a smaller representation of data, we may incur more false positives, needing more layers. As mentioned before, the 7-bit configuration falls near our defined threshold for creating more layers; because of that, we see the high FNR for this configuration. To better explore this phenomenon, we present in the fourth row an additional experiment for the 7-bit configuration using a threshold of 0.03 FNR for our cutoff on filter construction.

As the results show, we add 2 more layers, but the overall space does not increase as much. What is more interesting is that in this configuration, the FNR for the filter falls to 0.1%, a better result than all other configurations.

For the second table, where we see the increased FNR on the first and last of the selected results, we observe that for a higher FRR, size decreases as expected; however, if it is too relaxed, the number of false results will increase significantly and result in a larger filter size, because of the error.

7.1.5 Comparison with other works

To compare with other consent storage approaches, we ran a benchmark configuration of TPC-H with 1 million tuples on the CUSTOMER table and compared the space used on each approach. Purpose Scan(PRACIANO *et al.*, 2022) with Purpose Filter as the purpose storage, in the fixed size configuration of 5-bit per tuple id. Sieve (PAPPACHAN *et al.*, 2020) configured to query CUSTOMER and Hippocratic Databases with the extra attribute using the implementation from (KONSTANTINIDIS *et al.*, 2021). Figure 21 shows the comparison.

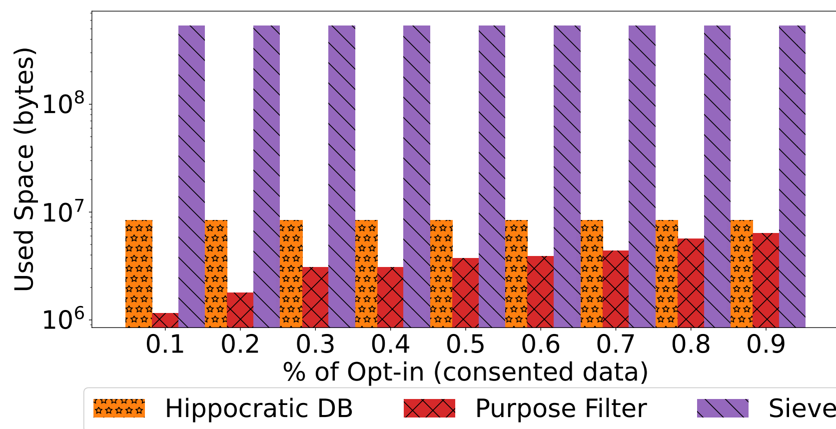


Figure 21 – Comparison with other works

The y-axis shows the storage space used in bytes for each approach, lower is better.

The x-axis shows the proportion of all the elements that are in the positive set. We observe that the other works have a fixed size, independent of the positive set size. Hippocratic DB (in orange and circles) stores less data than Sieve (purple, single slash), adding this overhead to query processing as added predicates that are evaluated upon query processing. Sieve constructs indexes auxiliary to data storage, increasing the amount of metadata stored.

We can see that, depending on the positive set size, our approach (red, cross-hatched) requires about half an order of magnitude less storage than Hippocratic Databases and almost 3 orders of magnitude less than Sieve, storing less than 10MB versus almost 1GB, highlighting the gain in storage size.

Although our approach scales size with the positive set size, we see from the experiment that even with 90% of the consents being positive, we are still below Hippocratic DB, matching what was observed from our previous experiments, in Figure 19a.

Discussion From these experiments, we observe a trade-off between storage space and the completeness of the results. Although it is a characteristic of the filter, it can be established dynamically based on the number of tuples and the size of the positive set. Considering scenarios where most of the Data Owners consent to their data being used to fully use a service, it is reasonable to expect that the positive set be the majority. Even with the highest level of FNR, 95.9% of the positive results are returned, which, in absolute numbers, are $6,713,000$ tuples out of $7,000,000$, which should not bring drawbacks to OLAP processing, being composed of aggregates and machine learning, as there is a small amount of data being excluded.

We also point out that these experiments were executed using a 5% threshold to stop adding layers to the filter. If the low FNR is more important, this threshold can be lowered, although a little more storage space will be used because of the extra layers.

We also identify that the threshold is as important to the resulting filter size and performance. By only changing the threshold in our experiment, we achieved a far better FNR by increasing our filter size by 4%. Depending on the application, this trade-off can be highly exploited.

7.2 Differentially Private Counting Bloom Filters

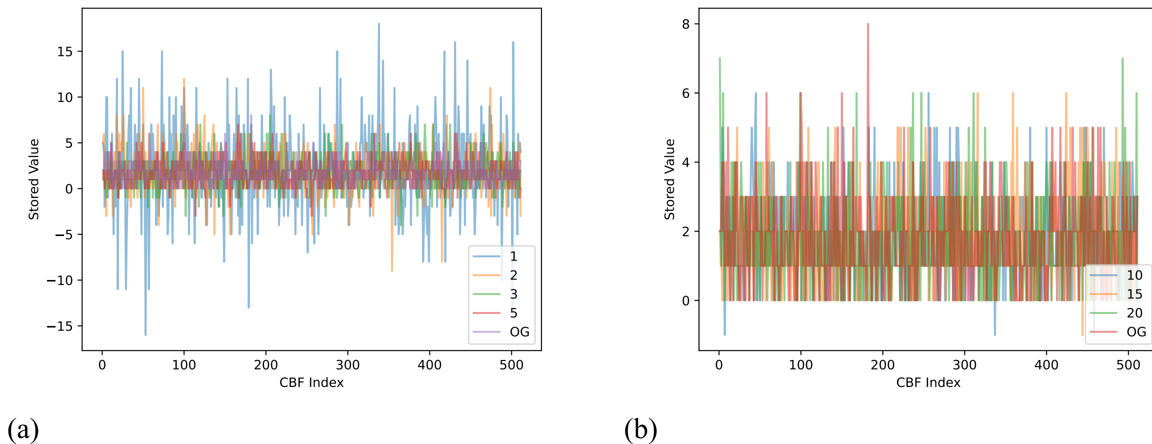
In this section, we present the experiments conducted with DPCBF. We discuss the setup choices, mainly selected to provide a direct comparison with DPBloomFilter (KE *et al.*, 2025). We vary our ε from 1 to 32, which is the range used in the DPBloomFilter paper.

We begin by showing a representation of the perturbed filter, comparing it to the unperturbed version, by showing the value in each index of the filter in a graph. Then, we perform our comparison to DPBloomFilter, analyze how the set reconstruction attack performs with different configurations and ε values, and finally, we show how the DPCBF can be used in a polling application, comparing the error with a non-perturbed version of the filter. Every experiment was run 5 times, and the average of the runs is presented in the results.

7.2.1 DPCBF Shape

To provide a better visualization and understanding of the added noise, figures 22a and 22b show the original filter values, labeled OG, and the noisy filters for each ε. The x-axis represents a CBF position, and the y-axis represents the value stored in that position. In figure 22a, we can observe that for ε values 3 and 5, the values start to be similar to the OG, and this is further observed in figure 22b, where the magnitude of values is much closer to the real ones.

Figure 22 – DPCBF shape for different values of ε



(a) (b)

Source: elaborated by the author.

7.2.2 Comparison with DPBloomFilter

In this experiment, we evaluate how the added geometric noise impacts the DPCBF with respect to false positives and false negatives, performing a comparison with DPBloomFilter and its Randomized Response. Our setup reflects the setup by DPBloomFilter, with the same defaults. By varying the three main components of the CBF, we can evaluate how the number of false results behaves. We use as default values $k = 3$ hash functions, the inserted set size

$|P| = 100000$, and the filter size $m = 2^{19}$. Recall that the added noise in each position is $\varepsilon' = \frac{\varepsilon}{k}$. The universe size is $500,000$.

We conduct the experiments by randomly generating 500,000 numbers, collecting 100,000 numbers to be inserted in the filters, executing the Constructor function in 3, and testing all 500,000 numbers to obtain the number of false positives and false negatives. All graphs are presented with the x-axis representing the value of ε, ranging between {1...32}, and the y-axis the absolute value of each, false negative (left) or false positives (right). Observe that in some graphs, the false positives have a black line across. This line represents the value of false positives on the non-perturbed filter. For false negatives, this value is always 0. Since we are dealing with misclassifications, all graphics should be read as: lower is better.

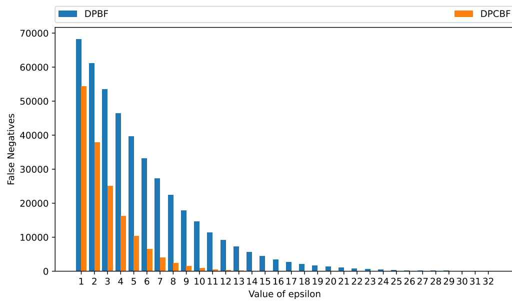
7.2.2.1 Varying k

Figures 23, 24, 25, 26, 27 show how the false positives and false negatives vary with 5 different values of $k = \{2, 3, 4, 5, 6\}$. As we observe in all graphics, the false negative amount decreases rapidly as ε increases, more so for the DPCBF, showing its superiority in comparison with DPBF with respect to false negative handling. Now, we discuss false positives.

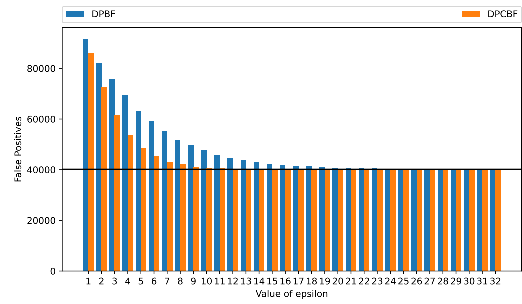
For values $k = 2$ and $k = 3$, both data structures converge to the original false positive quantity, although DPCBF converges faster, like in the false negative scenario.

For values of k over 3, we have the behavior of the false positives increasing instead of decreasing, as they begin below the original false positive. This happens because of two distinct factors. One is that we are actively increasing noise by increasing k. Do remember that our ε' is inversely proportional to k in its formula; therefore, as we increase k, for the same value of ε, we are adding more noise, because ε' is smaller. The other factor is more relevant to the data structure. When we increase k, what it means is that we are mapping an element to more positions in the filter. For the Bloom Filter, this may result in more 1 positions, but the collisions are overlooked, meaning that if the mechanism flips a 1 position, it may be influencing more than one element. In the Counting Bloom Filter, possible collisions would increase the value on a given position, making it more resistant to change (i.e., the added geometric noise would have to be negative and greater than the value stored in that position). This explains why our filter converges quickly to the original value.

Also note that the total error for DPCBF is smaller because of how fast the false negatives decrease.

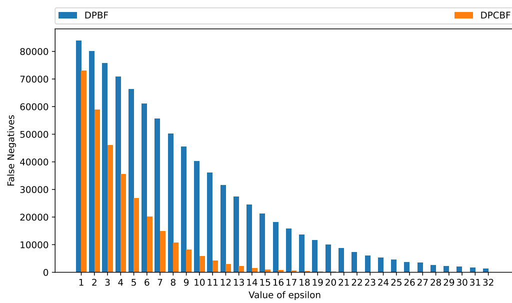


(a) False negatives

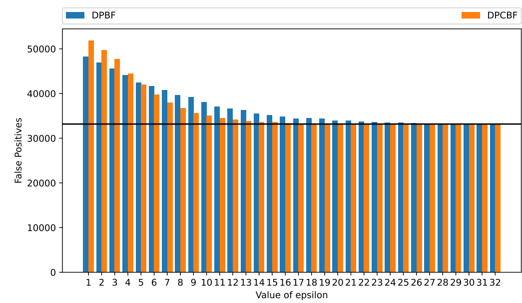


(b) False positives

Figure 23 – $k = 2$

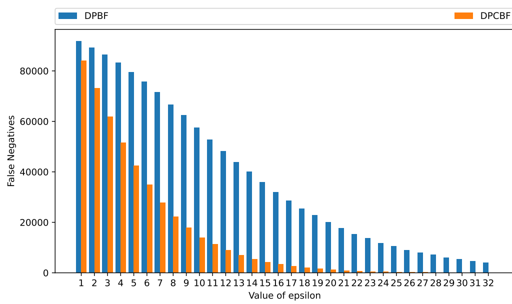


(a) False negatives

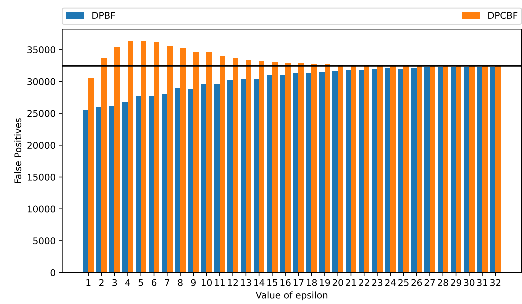


(b) False positives

Figure 24 – $k = 3$

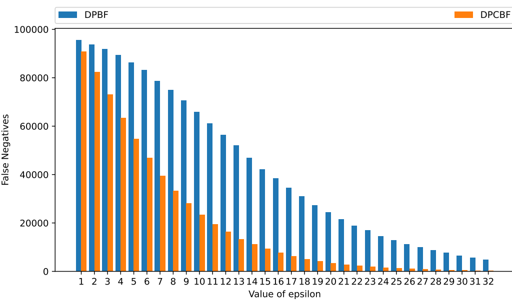


(a) False negatives

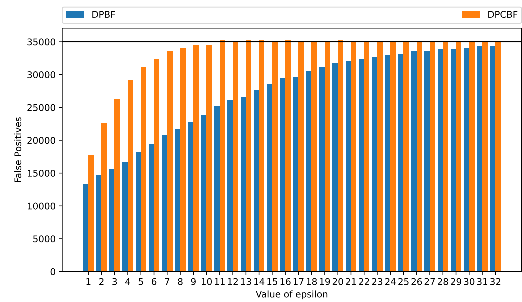


(b) False positives

Figure 25 – $k = 4$

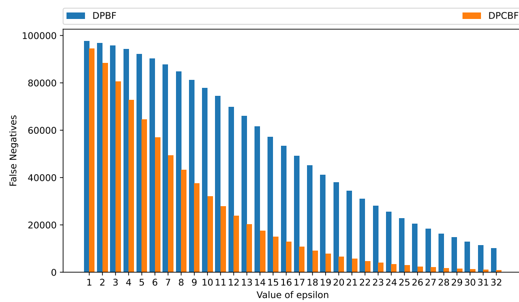


(a) False negatives

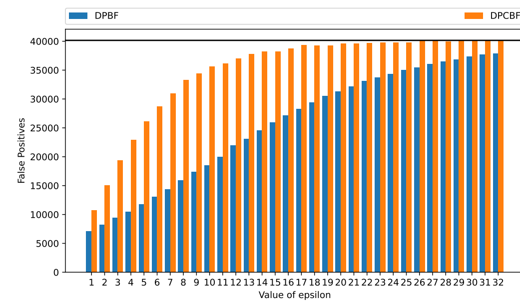


(b) False positives

Figure 26 – $k = 5$



(a) False negatives



(b) False positives

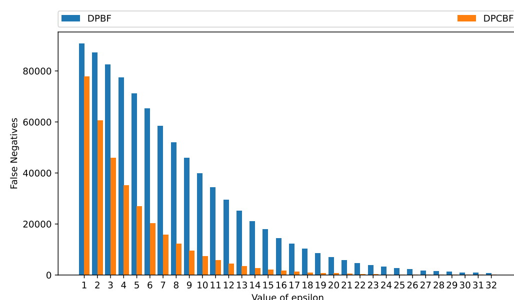
Figure 27 – $k = 6$

7.2.2.2 Varying m

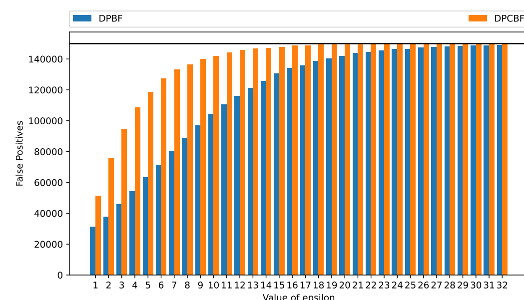
Figures 28, 29, 30, and 31 show the variation of false positives and false negatives as m takes values $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. In Figure 28b, we see a similar behavior to when we have many hash functions. While the amount of added noise is not impacted, the smaller filter size accounts for collisions, which increase the amount of false positives, even in the non-perturbed filter, as the black bar shows.

We see that for false negatives, in 28a to 31a that the DPCBF error (orange) decreases much faster than the DPBF (blue), highlighting the usefulness of DPCBF represented data over DPBF.

For higher sizes of m, we have fewer collisions, since the filter is larger, and the amount of false positives and negatives decreases. The DPCBF converges to the original value faster than DPBF, highlighting the increased utility of data when using DPCBF as the data structure.

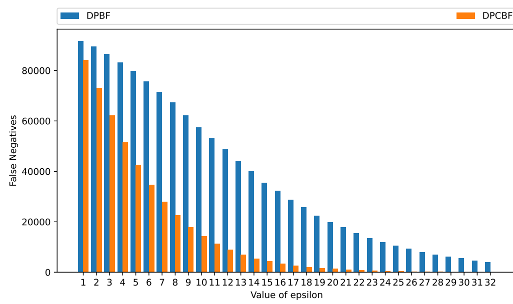


(a) False negatives

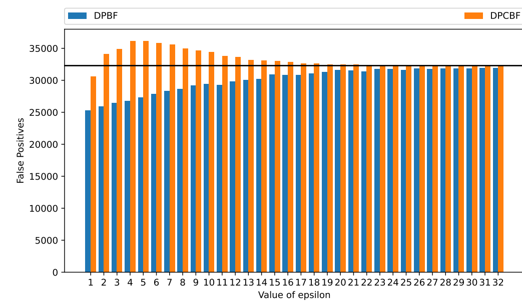


(b) False positives

Figure 28 – $m = 2^{18}$

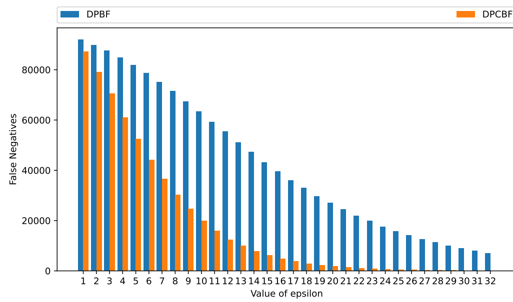


(a) False negatives

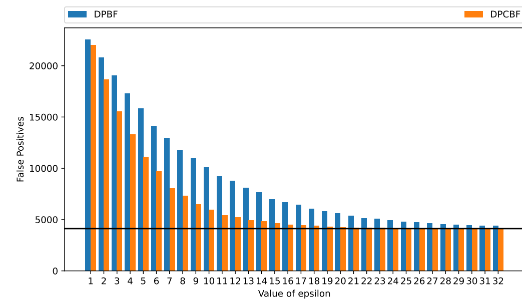


(b) False positives

Figure 29 – $m = 2^{19}$

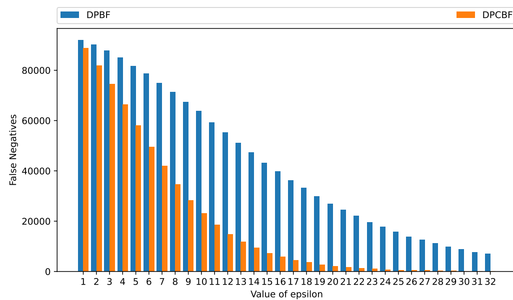


(a) False negatives

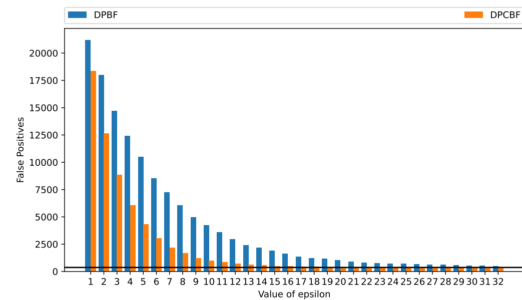


(b) False positives

Figure 30 – $m = 2^{20}$



(a) False negatives



(b) False positives

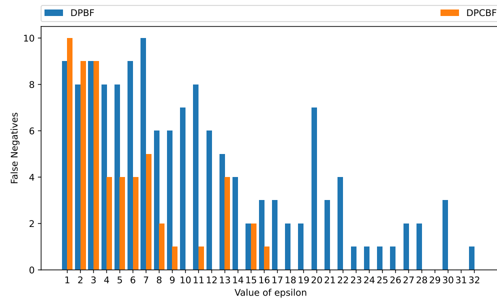
Figure 31 – $m = 2^{21}$

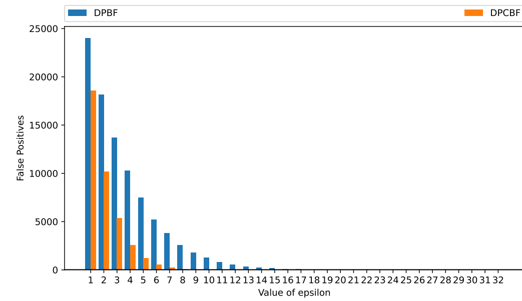
7.2.2.3 Varying $|P|$

Now we vary the inserted set size $|P|$ from 10 to 100,000, in figures 32, 33, 34, 35, and 36. In terms of false positives, we only see a change with $|P| = 100000$, because the quantity of added items makes it so that the filter does not have false positives with smaller amounts. Although it is a separate experiment, we can also see that figure 36b is similar to 29b, because the experimental setup, with $m = 2^{19}$, $|P| = 100000$, and $k = 3$ coincides.

For false negatives, we see that the DPCBF converges to the original value faster than DPBF, even in a scenario with only 10 possible items (figure 32a). As the number of items increases, the amount of false negatives decreases way faster than the DPBF, showing the utility

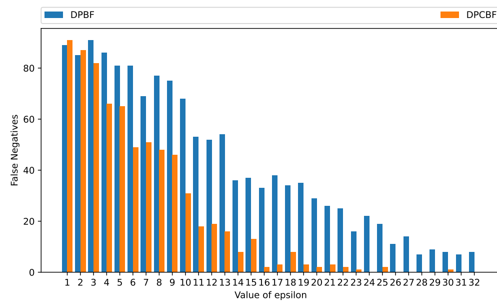
of DPCBF in this scenario.

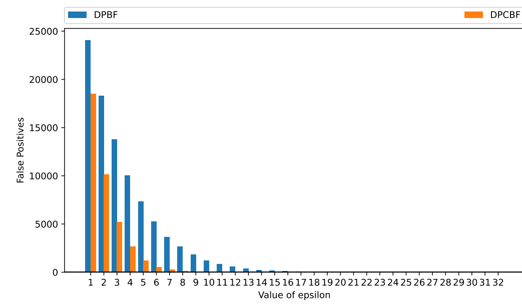




(a) False negatives (b) False positives

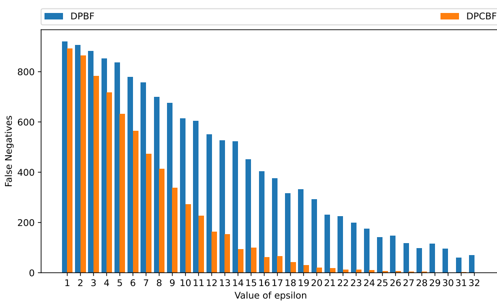
Figure 32 – Inserted size $|P| = 10$

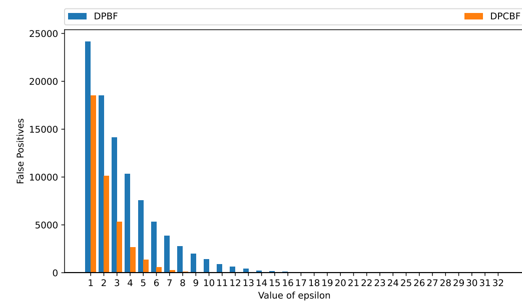




(a) False negatives (b) False positives

Figure 33 – Inserted size $|P| = 100$



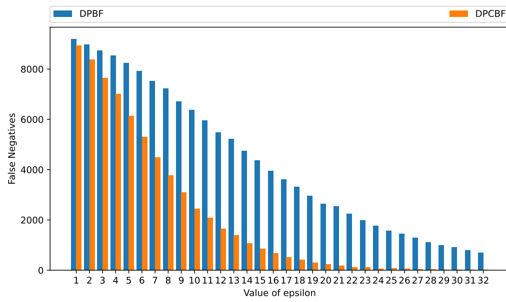


(a) False negatives (b) False positives

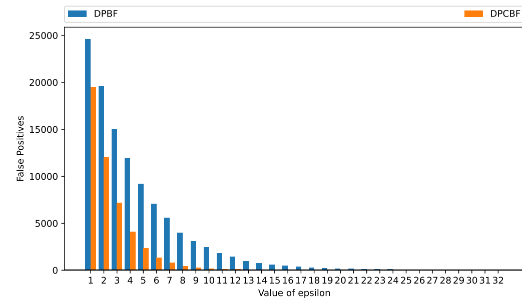
Figure 34 – Inserted size $|P| = 1000$

7.2.2.4 Discussion

From the experiments, we verify that for small values of ϵ, the performance of the filter is not significantly impacted for the task of classifying negatives. However, lower values of ϵ may destroy the filter capability of classifying positives, meaning that, depending

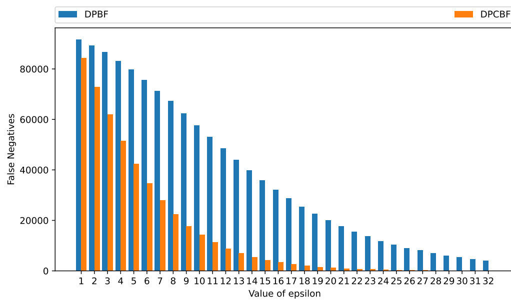


(a) False negatives

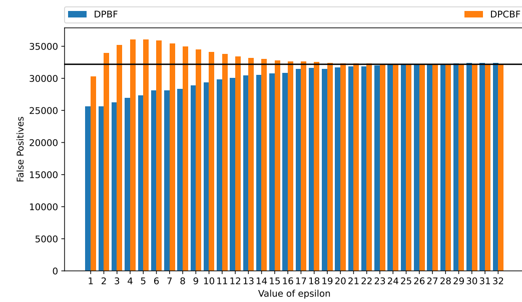


(b) False positives

Figure 35 – Inserted size $|P| = 10000$



(a) False negatives



(b) False Positives

Figure 36 – Inserted size $|P| = 100000$

on what application the filter is used in, more privacy budget must be provided to keep the filter useful. Nevertheless, our DPCBF is able to provide a better performance than DPBF, not only in the false negative scenario, but also in the false positive, provided an appropriate filter configuration (values of k and m according to $|P|$), with a lower error to the same value of ε. In the next experiment, we will evaluate the protection of different ε values in the context of the reconstruction attack from (REVIRIEGO *et al.*, 2023b).

7.2.3 Attack resistance

In this experiment, we show how the DPCBF is resistant to the reconstruction attack presented in (REVIRIEGO *et al.*, 2023b). Using the setups from the work, we use the Jaccard similarity to measure the similarity between the original set and the reconstructed set. The Jaccard similarity is defined as: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, meaning that the closer to 1, the more similar the sets are.

Tables 4, 5, 6 show the evolution of the Jaccard similarity as we vary the ε value. Every experiment was executed with a setup that reconstructed 100% of the inserted set in the original work.

Table 4 shows the Jaccard similarity for different ε values, keeping the value of $m = 65536$ counters and varying k. $|P|$ was adjusted to be the optimal value for a given k.

Table 4 – Jaccard similarity for different ε values and constant $m = 65536$.

ε	1	5	10	15	25
$k = 3$ and $\lvert P\rvert = 14780$	0.06402	0.23637	0.47186	0.68968	0.96109
$k = 4$ and $\lvert P\rvert = 11200$	0.04189	0.17803	0.39297	0.63276	0.94827
$k = 5$ and $\lvert P\rvert = 9000$	0.02601	0.13803	0.32839	0.56262	0.89546
$k = 6$ and $\lvert P\rvert = 7500$	0.01631	0.10648	0.27929	0.49565	0.84556
$k = 7$ and $\lvert P\rvert = 6450$	0.00944	0.08042	0.23006	0.43527	0.78038
$k = 8$ and $\lvert P\rvert = 5650$	0.00548	0.05730	0.18562	0.37011	0.72568

In table 4 we observe that, depending on the configuration, even for somewhat high values of ε, the Jaccard similarity is still low, meaning that the reconstruction attack is not able to reconstruct the original set. For example, for $\varepsilon = 10$ and $k = 3$, the Jaccard similarity is 0.47186, meaning that 47% of the original set was reconstructed, less than half. As we increase ε, the Jaccard similarity increases, but it does not reach 1, meaning that the attack is not able to reconstruct the original set completely. All configurations yield a similar number of hits in the filter, approximately 42,000. However, with higher values of k, the number of hits in the filter does not come from the number of elements inserted, meaning that it is easier to miss an element. This behavior is seen when we increase k and the Jaccard similarity decreases.

Table 5 shows the Jaccard similarity for different ε values, using different configurations of m and $|P|$, keeping a constant proportion between them. The values of k were kept constant at 5, and as mentioned, the original attack was able to reconstruct 100% of the inserted set ($JS = 1$).

Table 5 – Jaccard similarity for different ε values and constant $\frac{|P|}{m}$.

ε	1	5	10	15	25
$m = 1024$ and $\lvert P\rvert = 141$	0.02919	0.17590	0.37976	0.65219	0.93804
$m = 8192$ and $\lvert P\rvert = 1125$	0.02517	0.15769	0.35634	0.59077	0.92790
$m = 65536$ and $\lvert P\rvert = 9000$	0.02691	0.13741	0.32865	0.55984	0.89871
$k = 524288$ and $\lvert P\rvert = 72000$	0.04047	0.28663	0.60007	0.81449	0.97137

In this experiment, we have a similar ”occupation rate” of the filter, which is 68%. Meaning that after insertion, 68% of filter positions are not 0. For less items, in line 1, we have the Jaccard similarity a bit higher than other configurations besides the last. This happens because the filter size is too small, meaning that we don’t have as many chances to introduce noise. For

the other configurations, behavior is as expected; as we add more items, the Jaccard similarity increases, even though the filter size also increases.

Table 6 shows the Jaccard similarity for different ε values, varying k by $\{3, 4, 5\}$, keeping $m = 65536$ and changing $|P|$ between the 0.5x optimal value, 1x the optimal value and 1.5x the optimal value.

Table 6 – Jaccard similarity for different ε values and constant $m = 65536$.

ε	1	5	10	15	25
$k = 3$ and $\|P\| = 7435$	0.04290	0.25253	0.65432	0.90835	0.99561
$k = 3$ and $\|P\| = 14870$	0.06402	0.23637	0.47186	0.68968	0.96109
$k = 3$ and $\|P\| = 22305$	0.08297	0.24287	0.08555	0.06340	0.05988
$k = 4$ and $\|P\| = 5600$	0.02944	0.18659	0.56531	0.84130	0.98576
$k = 4$ and $\|P\| = 11200$	0.04189	0.17803	0.39297	0.63276	0.94827
$k = 4$ and $\|P\| = 16800$	0.05121	0.17865	0.32191	0.45292	0.68669
$k = 5$ and $\|P\| = 4500$	0.01881	0.13958	0.46069	0.75136	0.96227
$k = 5$ and $\|P\| = 9000$	0.02601	0.13803	0.32839	0.56262	0.89546
$k = 5$ and $\|P\| = 13500$	0.03239	0.14062	0.26546	0.39678	0.68376

As we increase the number of items in a given filter configuration, we increase the number of collisions and the false positives. This is reflected in the Jaccard similarity, which decreases even on the last column of each k value. Besides the added noise, we have collisions, which increase the false positive rate. A very interesting result is in the third line, as we have many items added, and the Jaccard similarity drastically decreases, even with high values of ε. Since there are too many items in the inserted set, we attribute the low similarity to the inability of the reconstruction attack to find the original set, because of the inserted noise, even if it was a small amount. As an odd result, we reran the experiments additional times and confirmed this result.

7.2.4 Application: Polling

To provide an application example for DPCBF, we present a polling scenario where we have two different candidates, and we want to know how many people voted for each candidate, without revealing who voted for whom, providing a privacy guarantee through the data structure used to store the votes. If we use a regular Counting Bloom Filter to store the votes, we can add up the value of all positions in the filter, and divide by the quantity of hash functions to know how many people voted a given candidate, but we cannot guarantee that the votes are private, as the Counting Bloom Filter is not differentially private.

We set up the experiment with 1000000 voters, and two candidates, A and B. Each voter votes for one of the candidates, and we store the votes in a Counting Bloom Filter, one for each candidate. We split our voters by random drawing, but keep the proportions 60-30-10, meaning that around 60% voted for A, approximately 30% voted for B, and the rest 10% did not vote for either.

The filters are set up with $m = 5000000$ and $k = 5$, to handle whatever proportion of voters, even if everyone would vote for the same candidate. While this provides more chances to increase noise, since we have more positions, we can still guarantee that the Counting Bloom Filter is able to handle any scenario.

To show the utility of the DPCBF in this scenario, we used lower values of ε, namely $\varepsilon = \{0.1, 0.5, 1, 2, 5, 10\}$. Although a lower value of ε than our previous experiments, in here, we are interested in the sum of the values in the filter, much like a histogram, and using lower values of ε provides more privacy to individuals, in terms of queries, as the reconstruction attack uses. The original vote setup does not change, but we run each experiment 10 times for each value of ε, and calculate the Mean Absolute Error (MAE) between the original votes and the votes counted from the DPCBF. We define MAE as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |x_i - y_i| \tag{7.1}$$

Where x_i is the original vote count for candidate i, and y_i is the counted votes from the DPCBF. The results are shown in Table 7.

Table 7 – MAE for the polling scenario with different ε values.

ε	MAE Candidate A	MAE Candidate B
0.1	6925	14892
0.5	1769	1929
1	1455	675
2	656	405
5	182	236
10	109	92

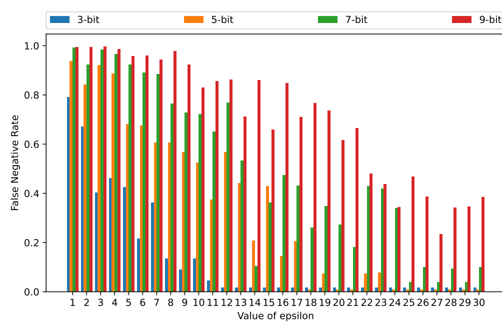
From these results, we can see that the DPCBF is able to provide a good approximation of the votes, even with low values of ε. As shown in previous experiments, the reconstruction attack has a Jaccard similarity of less than 0.05 for $\varepsilon = 1$, and the error from the MAE is representative of less than 0.5% of the voters, falling very well into an error margin for a polling scenario. This shows promising utility for using the filter as an alternative to histograms, where we can guarantee that the votes are private, and the results are still useful for the polling scenario.

7.3 Differentially Private Purpose Filter

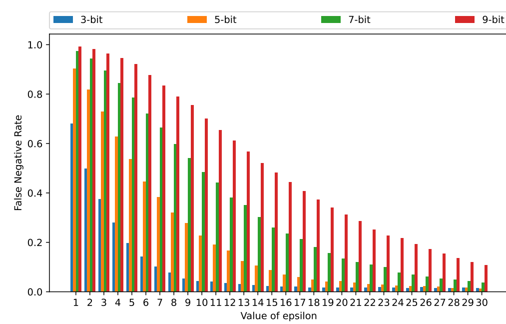
In this section, we run experiments comparing two types of layers for Purpose Filter construction, Bloom Filters and Counting Bloom Filters, both made differentially private using the algorithms of DPBloomFilter and DP-Counting Bloom Filter, comparing the false negative rate (FNR) in different configurations, according to the experiments run in section 7.1.

7.3.1 Comparison of layer types

Figures 37, 38, 39, 40, and 41 show the variation of FNR for Purpose Filter when we turn it differentially private. We compare two layer types, Bloom Filter and Counting Bloom Filter. Although in a non-perturbed scenario, this comparison is not relevant, as both yield the same result on query, when we add noise, the Counting Bloom Filter is able to provide a better result, as it is able to handle false negatives better than the Bloom Filter, because of the type and amount of added noise. We ran the same experiment as 7.1.2, now perturbing the first layer using the algorithm from DPBloomFilter, for the Bloom Filter, and the algorithm from DPCBF, for the Counting Bloom Filter. We perform this experiment with $k = \{3, 5, 7, 9\}$ and the filter size being $(k * |P|)$. We ran the experiments varying $|P|$ from 5% to 95% of the universe size (100000), and varying ε from 1 to 30. To better present the results, we selected 5 configurations for $|P|$, $|P| = \{10000, 30000, 55000, 70000, 90000\}$.



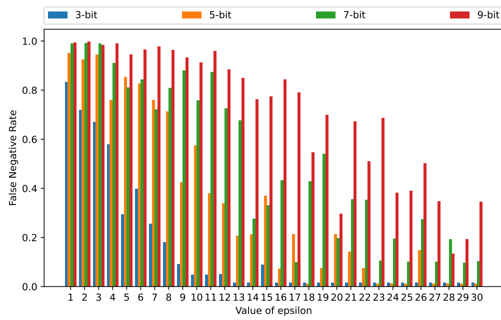
(a) DPBloomFilter



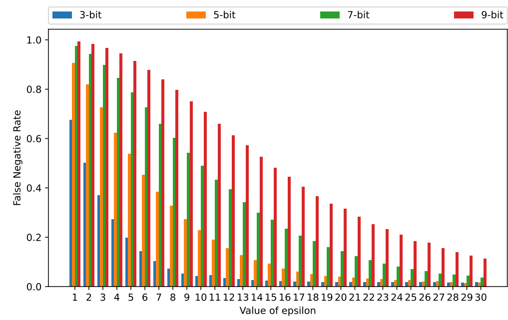
(b) DPCBF

Figure 37 – Inserted size $|P| = 10000$

From the previous experiments, we know that a non-perturbed Purpose Filter is resistant to the change in positive set size proportion (refer to 19b). For the perturbed filter, this characteristic remains, as can be seen by comparing all figures from DPBloomFilter and DPCBF. However, the noise introduction highly impacts the false negative rate of DPBloomFilter, having

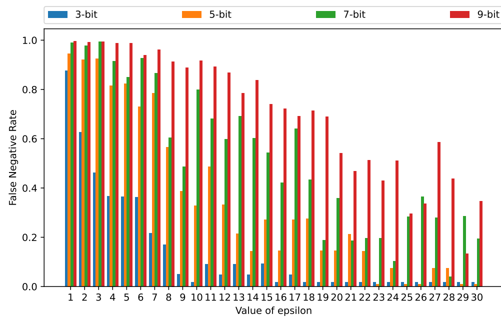


(a) DPBloomFilter

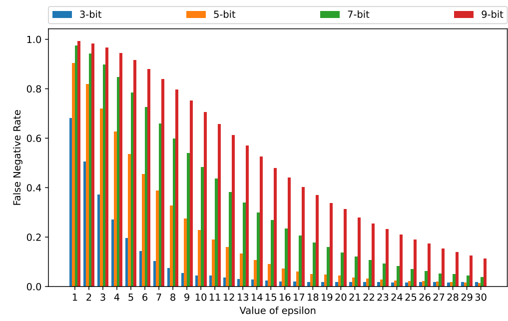


(b) DPCBF

Figure 38 – Inserted size $|P| = 30000$

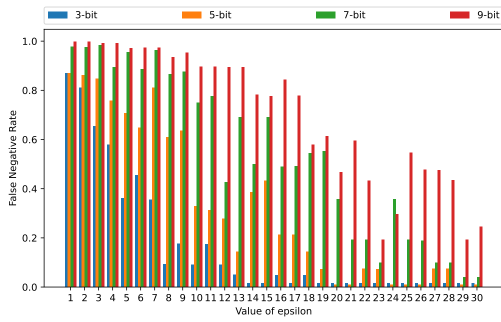


(a) DPBloomFilter

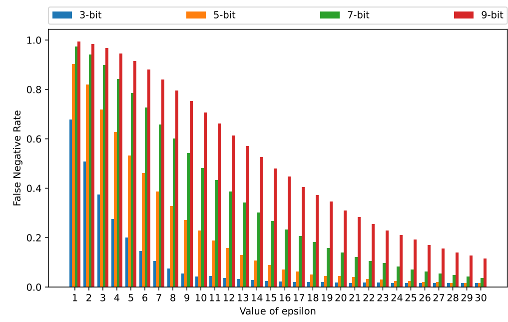


(b) DPCBF

Figure 39 – Inserted size $|P| = 55000$

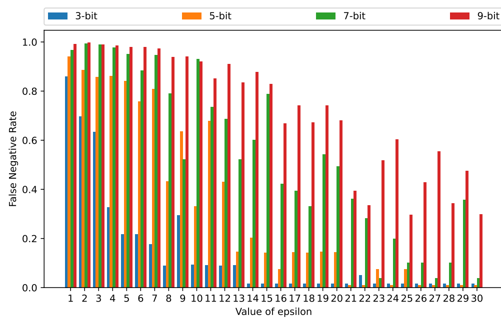


(a) DPBloomFilter

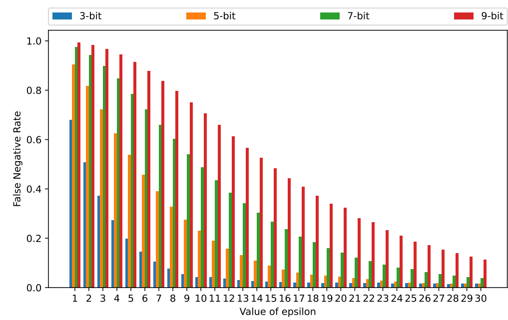


(b) DPCBF

Figure 40 – Inserted size $|P| = 70000$



(a) DPBloomFilter



(b) DPCBF

Figure 41 – Inserted size $|P| = 90000$

a very high FNR even with high values of ε. The DPCBF, on the other hand, is more tame and gracious with the noise. With $\varepsilon = 1$, the DPBloomFilter version goes from 80% FNR to 100% FNR depending on the configuration, while the DPCBF starts from around 64%, and converges faster to a low value, as can be seen with lower hash sizes.

The quantity of positions increases with the number of hash functions, further amplifying the effect of the perturbation, which is why the 9-bit configuration is much higher than the other configurations. For lower amounts of hash functions, we achieve a low FNR even with a not-so-high value of ε.

The addition of noise highly impairs the utility of data, especially when the noise is very high (low values of ε). Nevertheless, we have good protection even with high values of ε, as the attack experiment shows, meaning that we can somewhat afford a looser bound on added noise, and as this experiment shows, with a configuration of 3 hashes per element, we go to less than 10% of false negatives with $\varepsilon = 8$, which in a similar configuration, yields a Jaccard similarity of less than 0.47, showing that the attacker's probability of correctly ascertaining if an element is in the set is less than that of a fair coin flip.

7.4 Application: ML training

In this section, we present an application of consent enforcement in the context of machine learning training. To verify the impact of consent providing and our data structures, both Purpose Filter and DP-Purpose Filter, we executed the same training and test over 4 different datasets. A complete dataset, a consented dataset where 90% of the users accepted their data to be used, therefore, we remove 10% of the training data; the same consents stored in Purpose Filter, where we remove the 10% non-consented data plus the false negatives that it may produce; and finally, we use DP-Purpose Filter to store the consents, knowing that the false negative rate is higher because of the added noise. From our previous experiments, we understand that using Counting Bloom Filters as layers, and its DP counterpart provides a better false negative rate; therefore, that is how we set up the DP-Purpose Filter.

7.4.1 Application and dataset

To work with a realistic dataset that we would be able to evaluate the impact of consent providing through an objective metric, and use a known workflow to do so, we used

the APS dataset from Scania (APS…, 2017). The choice was motivated by both the number of instances (60000), because it is a real-world dataset, and it was used in a competition in the Industrial Challenge 2016 at The 15th International Symposium on Intelligent Data Analysis (IDA), providing a known workflow to follow and a suitable metric to evaluate the results.

The training dataset is composed of 60000 instances, with 171 features, and a target variable that is a binary classification, where the goal is to predict if the target variable is *pos* or *neg*. A test dataset with 16000 instances is also provided. The failures are weighed, meaning that the cost of a false negative is higher than the cost of a false positive, because the goal of this task is to detect problems in the air pressure system of the trucks, which impacts braking and transmission. A false positive means that the truck is working properly, but was sent to be repaired, which spends time, but isn't that costly, requiring only the diagnosis to be corrected. A false negative means that the truck is not working properly, but was not sent to be repaired, which can cause a lot of problems, as the truck may break down in the middle of the road, and require the whole system to be replaced. In ML terms, the goal is to minimize a given cost, which is calculated as:

$$Cost = 10 * FP + 500 * FN \tag{7.2}$$

Where FP is the number of false positives, and FN is the number of false negatives.

These datasets are highly imbalanced, as expected. The number of faulty trucks is small; in the training dataset, there are 1000 faulty trucks and 59000 healthy trucks. The dataset also has missing data in some of its registers.

7.4.2 Setup and workflow

For our experiments, we set up 4 scenarios regarding the training dataset. We did not alter the test dataset to keep the results comparable, and we used the same test dataset for all scenarios. The scenarios are:

- Complete dataset: The complete training dataset, with all 60000 instances.
- Consented dataset: The training dataset with 90% of the instances, meaning that we remove 10% of the instances, which are randomly selected.
- PF consented dataset: The training dataset with 90% of the instances, but we use a Purpose Filter to store the consents, meaning that we also have false negatives, which remove other

Table 8 – Exploratory analysis of the datasets.

Dataset	Instances	Positives	Negatives
Complete	60000	1000	59000
Consented	54000	886	53114
PF consented	53460	877	52583
DPPF consented	43705	734	42971

instances from the training dataset.

– DPPF consented dataset: The training dataset with 90% of the instances, but we use a DP-Purpose Filter to store the consents, increasing the number of false negatives, as previous experiments showed.

For all consented datasets, the consents are the same, to preserve the results' integrity. For the filters, we used $k = 5$ and $m = k * n$, where n is the number of instances in the training dataset. For the DPPF, we used an $\varepsilon = 11$, to have some utility in the filter and provide a good protection against the reconstruction attack.

The ML workflow is:

– We perform some exploratory analysis of the datasets.
– We perform cross-validation splits
– We perform data imputation, as the dataset has some missing values. The imputation is done using the Mean, Median, and Mode.
– We train 4 different models with imbalanced data: LogReg, SVM, Random Forest, and XGBoost.
– We oversample data to balance the dataset, using SMOTE, and then train Random Forest and XGBoost with this new dataset.
– We evaluate the models using the test dataset and calculate the cost of each model.

7.4.3 Exploratory analysis

We have 4 different training datasets, and we perform exploratory analysis on each of them. Table 8 shows the characteristics of each dataset.

From the table, we can see that we lost almost 25% of the dataset when using the DPPF, which is expected, as we are adding noise to the filter, and therefore increasing the false negative rate. The false negatives on the plain Purpose Filter are almost negligible. However, even though we lost many instances, the imbalance of the original dataset is almost preserved. If we check proportionally, the complete dataset has 1.67% of positives, the consented dataset has

1.64%, the PF consented dataset has 1.64%, and the DPPF consented dataset has 1.68%. As we will see in the results, we get reasonably close in all datasets.

7.4.4 Results

Tables 9 to 11 show the results for each configuration, without oversampling, and in bold is the best test score. Training scores are added to help visualize any sort of overfitting.

Table 9 – Results for mean imputation

Dataset	Train Cost	Test Cost
Complete	29710	**15070**
Consented	26070	18870
PF consented	25530	15990
DPPF consented	19730	22090

(a) Logistic. Reg.

Dataset	Train Cost	Test Cost
Complete	29710	13770
Consented	26070	**13690**
PF consented	25530	16120
DPPF consented	19730	16060

(b) SVM

Dataset	Train Cost	Test Cost
Complete	50020	16160
Consented	51130	18240
PF consented	10610	12970
DPPF consented	21150	**12620**

(c) Random Forest

Dataset	Train Cost	Test Cost
Complete	14730	**9630**
Consented	7780	12960
PF consented	7570	13890
DPPF consented	11170	10720

(d) XGBoost

For the mean imputation (tables 9a to 9d), we observe that the lack of data on the other datasets impacts classic models such as Logistic Regression and SVM. We see some overfitting as data decreases, given the decreasing training cost and increasing test cost. For Random Forest, the reduced dataset greatly helped the classification results. For XGBoost, the more representative mean on the complete dataset yielded the best result, although DPPF had a better score than the other consent datasets.

For the median imputation (tables 10a to 10d), the DPPF version of the dataset greatly suffered, mainly because the median values for this dataset are very far from the median in others. Remember that the DPPF dataset has less than 75% of the original dataset. However, the XGBoost classifier did work well with this dataset, while the complete yielded the best result, DPPF was the second place in this experiment.

Table 10 – Results for median imputation

Dataset	Train Cost	Test Cost
Complete	29750	**14140**
Consented	25240	18100
PF consented	22390	16530
DPPF consented	19100	21680

(a) Logistic Reg.

Dataset	Train Cost	Test Cost
Complete	36940	**13440**
Consented	33360	14990
PF consented	31110	13920
DPPF consented	25080	17490

(b) SVM

Dataset	Train Cost	Test Cost
Complete	20100	11000
Consented	14320	11140
PF consented	18250	**10270**
DPPF consented	44550	20310

(c) Random Forest

Dataset	Train Cost	Test Cost
Complete	27900	**9560**
Consented	4760	14360
PF consented	13930	11690
DPPF consented	8540	11640

(d) XGBoost

Table 11 – Results for mode imputation

Dataset	Train Cost	Test Cost
Complete	32260	**14790**
Consented	26400	18560
PF consented	24580	17450
DPPF consented	20920	22380

(a) Logistic Reg.

Dataset	Train Cost	Test Cost
Complete	39520	13560
Consented	34810	16870
PF consented	31120	**13200**
DPPF consented	25260	15340

(b) SVM

Dataset	Train Cost	Test Cost
Complete	27450	11090
Consented	16730	**10510**
PF consented	45700	16310
DPPF consented	35250	14820

(c) Random Forest

Dataset	Train Cost	Test Cost
Complete	7170	**11960**
Consented	5890	14110
PF consented	2860	16280
DPPF consented	6430	12650

(d) XGBoost

For the mode imputation (tables 11a to 11d), we do have good results in Random Forest and XGBoost. These results show that while we have an impact from removing registers from the dataset because of consents/false negatives, the overall performance of the models is not completely impacted. Now we verify our results using oversampling to compensate for the skewed dataset.

We then perform the experiments after applying the Synthetic Minority Oversampling Technique (SMOTE) before training. For this configuration, we used only tree-based models (Random Forest and XGBoost). Tables 12 to 14 show the results for each configuration, with SMOTE:

Table 12 – Results for mean imputation/SMOTE

Dataset	Train Cost	Test Cost
Complete	5870	12660
Consented	10270	9950
PF consented	15010	**9200**
DPPF consented	10150	10070

(a) Random Forest

Dataset	Train Cost	Test Cost
Complete	14510	10190
Consented	15700	16460
PF consented	20070	**9090**
DPPF consented	25960	11890

(b) XGBoost

Table 13 – Results for median imputation/SMOTE

Dataset	Train Cost	Test Cost
Complete	4870	11470
Consented	10660	**8900**
PF consented	13690	10490
DPPF consented	13480	9860

(a) Random Forest

Dataset	Train Cost	Test Cost
Complete	3970	15300
Consented	4520	15460
PF consented	15260	**9360**
DPPF consented	4330	13980

(b) XGBoost

Table 14 – Results for mode imputation/SMOTE

Dataset	Train Cost	Test Cost
Complete	5260	13150
Consented	15620	12260
PF consented	14650	**9850**
DPPF consented	5460	14710

(a) Random Forest

Dataset	Train Cost	Test Cost
Complete	15260	10580
Consented	7720	13100
PF consented	15150	**9950**
DPPF consented	13380	10170

(b) XGBoost

With oversampling, we see an increase in performance of the model, and since we had a similar proportion of positives (approximately 1.67% of the dataset) over all datasets, fewer negatives helped the model to behave better, as all consented datasets improved over the complete dataset. For median imputation, the same happens, although the PF consented dataset is the second-worst with Random Forest and the best with XGBoost. Finally, for mode imputation, the PF consented dataset performed the best, but overall, all results were close enough.

This application shows that, although there is an impact on data due to consents, and even additional data loss because of consent data protection, ML models and techniques are able to deal with it, especially more modern algorithms like Random Forest and XGBoost. The fact that data is highly imbalanced could present a threat to our model, if many instances of the underrepresented class were removed by false negatives in the filter, however, the data proportions were kept, without using any kind of selection or attempt to favor this class, showing that we were still capable of keeping the original characteristics of the dataset, even when removing some part of it.

8 CONCLUSION

In this chapter, we conclude our thesis, presenting a summary of the results and how they support our hypothesis, as well as providing directions for future work.

8.1 Summary of results

In this work, we addressed the problem of efficiently representing consent data and protecting it under differential privacy. We propose both a data structure that can improve storage and can be perturbed using differential privacy techniques. We propose an alternative algorithm that can be used to perturb a different kind of layer, and we show how to integrate this data structure, Purpose Filter, with differential privacy without violating consent constraints, Differentially Private Purpose Filter. Both became research papers. The Purpose Filter work became a full research paper, accepted into DbSec'25, and Differentially Private Purpose Filter is under editing for submission to a top-tier conference.

We perform a literature review on two fronts: Consent enforcement in databases and differential privacy in filters, as filters are the foundation of our data structure. We then focus on the privacy assumptions of filters, showing that their probabilistic nature is not enough to protect inserted items, which shows the need for further privacy, motivating the second part.

Let us revisit our research problem and hypothesis:

Research Problem: *Consider a set of identifiers U for a given purpose, split in opt-ins (P) and opt-outs (N). Our goal is to release a data structure that allows for publishing and set membership queries while ensuring protection to the identifiers inserted in this data structure and providing usefulness.*

Research Hypothesis: Given a universe of identifiers U, partitioned into opt-in (P) and opt-out (N) subsets such that $U = P \cup N$, one can construct a data structure that (i) enables useful representation and enforcement of consent, (ii) ensures privacy for members of both P and N, and (iii) achieves these guarantees while incurring low storage overhead relative to the size of U.

To prove our hypothesis true, we developed Purpose Filter to represent consent and allow for privacy mechanisms to be applied in the structure. To improve usefulness, we explored a different type of filter to compose its layers, and performed the necessary steps to turn this filter differentially private. To achieve our main objective, we needed to turn Purpose Filter

differentially private, through its layers. Nevertheless, the process is not straightforward, as direct noise addition to all layers could violate consent properties. Finally, by presenting how to achieve a Differentially Private Purpose Filter, we confirm our hypothesis.

Our experiments support these claims, starting from the analysis of Purpose Filter, showing that we achieve a low false negative rate, based on the configuration, and do not significantly impact storage overhead, compared to other approaches. We then moved to add differential privacy to our filter layers. Through our exploration, we designed a new differentially private filter: DP-Counting Bloom Filter (DPCBF), which performs better than differentially private Bloom filters (DPBF), demonstrating that we can improve the utility of differentially private filters. We also show that this filter is resistant to set reconstruction attacks and show an application of this filter to store poll results. Finally, we apply this filter as the top layer of Purpose Filter, ensuring privacy for the two sets used in filter construction and retaining usefulness, based on the filter configuration. We show that DPCBF works better than DPBF as layers of our differentially private Purpose filter (DPPF), yielding a lower false negative rate than the DPBF layer. We then test how these consent constraints affect a real-world application, using a machine learning scenario, with a real dataset and well-established metrics. We show that even though the dataset is reduced, the classifiers are still able to perform appropriately, depending on their configuration, showing that our private data retains usefulness, in comparison to the complete dataset.

8.2 Future Works

For future works, we acknowledge a more specific goal of enhancing purpose and consent representation and a more general goal regarding filters as a data structure and differential privacy.

For the first goal, it is necessary to investigate filter combinations and the usage of filters in a hierarchy. This type of usage is done in Bloom Lookup Tables, although in the context of pruning a search space. Assembling a set of filters as a hierarchy and allowing reuse of already built filters to construct new ones can improve purpose and consent representation, by increasing granularity (a filter composed of filters for each attribute, for example), and allowing a more general representation based on sets and subsets of purposes (modeling filters to represent an hierarchical set of purposes, e.g. Sales -> Food -> Dog treats), as this hierarchy can be used for better modeling of purposes and consents, from general to specific.

For the second goal, other types of filters are susceptible to set-reconstruction attacks, and the field of differential privacy for filters has so far concentrated on Bloom Filters. This means that other filters' distinct characteristics on how they encode data can be analyzed or exploited in a differential privacy manner, or, even better, since filters rely on hash functions to map their entries, one could design a differentially private hash function that would allow a degree of uncertainty in the map process. This could be applicable in a scenario where we could estimate or ascertain the mapping of the function based on the input, meaning these functions would not disperse data as well, but with this characteristic, it is possible to design a noise-adding mechanism to prevent attackers from determining the correct mapping of the function.

REFERENCES

AGRAWAL, R.; BIRD, P.; GRANDISON, T.; KIERNAN, J.; LOGAN, S.; RJAIBI, W. Extending relational database systems to automatically enforce privacy policies. **ICDE**, IEEE Computer Society, Tokyo, p. 1013–1022, 2005.

AGRAWAL, R.; KIERNAN, J.; SRIKANT, R.; XU, Y. Hippocratic databases. **VLDB**, Morgan Kaufmann, p. 143–154, 2002.

ALAGGAN, M.; GAMBS, S.; KERMARREC, A. BLIP: non-interactive differentially-private similarity computation on bloom filters. **SSS**, Springer, p. 202–216, 2012.

APS Failure at Scania Trucks. UCI-MLR, 2017. Accessed on 2025-07-13. Available at: https://doi.org/10.24432/C51S51.

BIANCHI, G.; BRACCIALE, L.; LORETI, P. "Better than nothing" privacy with bloom filters: To what extent? **Privacy in Statistical Databases**, Springer, p. 348–363, 2012.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. **Commun. ACM**, v. 13, n. 7, p. 422–426, 1970.

BONOMI, F.; MITZENMACHER, M.; PANIGRAHY, R.; SINGH, S.; VARGHESE, G. An improved construction for counting bloom filters. **ESA**, Springer, p. 684–695, 2006.

BYUN, J.; LI, N. Purpose based access control for privacy protection in relational database systems. **VLDB J.**, v. 17, n. 4, p. 603–619, 2008.

CCPA. **California Consumer Privacy Act**. 2018. Available at: https://oag.ca.gov/privacy/ccpa.

DEEDS, K.; HENTSCHEL, B.; IDREOS, S. Stacked filters: Learning to filter by structure. **Proceedings of the VLDB Endowment**, v. 14, n. 4, p. 600 – 612, 2021.

DESHPANDE, A. Sypse: Privacy-first Data Management through Pseudonymization and Partitioning . **CIDR**, p. 1–8, 2021.

DUCHI, J. C.; JORDAN, M. I.; WAINWRIGHT, M. J. Local privacy and statistical minimax rates. **2013 IEEE 54th Annual Symposium on Foundations of Computer Science**, p. 429–438, 2013.

DWORK, C. Differential privacy. **International colloquium on automata, languages, and programming**, p. 1–12, 2006.

DWORK, C.; MCSHERRY, F.; NISSIM, K.; SMITH, A. Calibrating noise to sensitivity in private data analysis. **Third Theory of Cryptography Conference**, p. 265–284, 2006.

DWORK, C.; ROTH, A. *et al.* The algorithmic foundations of differential privacy. **Foundations and Trends® in Theoretical Computer Science**, Now Publishers, Inc., v. 9, n. 3–4, p. 211–407, 2014.

ERLINGSSON, Ú.; PIHUR, V.; KOROLOVA, A. Rappor: Randomized aggregatable privacy-preserving ordinal response. **Proceedings of the 2014 ACM SIGSAC conference on computer and communications security**, p. 1054–1067, 2014.

FAN, B.; ANDERSEN, D. G.; KAMINSKY, M.; MITZENMACHER, M. Cuckoo filter: Practically better than bloom. **CoNEXT**, ACM, p. 75–88, 2014.

FREEDMAN, M. J.; NISSIM, K.; PINKAS, B. Efficient private matching and set intersection. **EUROCRYPT**, Springer, p. 1–19, 2004.

GALÁN, S.; REVIRIEGO, P.; WALZER, S.; SÁNCHEZ-MACIÁN, A.; LIU, S.; LOMBARDI, F. On the privacy of counting bloom filters under a black-box attacker. **IEEE Trans. Dependable Secur. Comput.**, v. 20, n. 5, p. 4434–4440, 2023.

General Data Protection Regulation. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46. **Official Journal of the European Union**, v. 59, p. 1–88, 2016.

GHOSH, A.; ROUGHGARDEN, T.; SUNDARARAJAN, M. Universally utility-maximizing privacy mechanisms. **Proceedings of the forty-first annual ACM symposium on Theory of computing**, p. 351–360, 2009.

GRAF, T. M.; LEMIRE, D. Xor filters: Faster and smaller than bloom and cuckoo filters. **CoRR**, 2019.

HUFFMAN, D. A. A method for the construction of minimum- redundancy codes. **Proceedings of the IRE 40**, v. 9, p. 1098–1101, 1952.

KE, Y.; LIANG, Y.; SHA, Z.; SHI, Z.; SONG, Z. Dpbloomfilter: Securing bloom filters with differential privacy. **CoRR**, 2025.

KONSTANTINIDIS, G.; HOLT, J.; CHAPMAN, A. Enabling personal consent in databases. **Proc. VLDB Endow.**, v. 15, n. 2, p. 375–387, 2021.

KRASKA, T.; STONEBRAKER, M.; BRODIE, M. L.; SERVAN-SCHREIBER, S.; WEITZNER, D. J. SchengenDB: A data protection database proposal. **Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019**, Springer, California, p. 24–38, 2019.

LGPD. **Lei Geral de Proteção de Dados**. 2018. Available at: http://www.planalto.gov.br/c-civil_03/_ato2015-2018/2018/lei/L13709compilado.htm.

MACHADO, J. C.; AMORA, P. R. P. The impact of privacy regulations on DB systems. **Journal Information Data Management**, v. 12, n. 5, 2021.

MITZENMACHER, M. Balanced allocations and double hashing. **SPAA**, ACM, p. 331–342, 2014.

PANDEY, P.; BENDER, M. A.; JOHNSON, R.; PATRO, R. A general-purpose counting filter: Making every bit count. **SIGMOD Conference**, ACM, Chicago, p. 775–787, 2017.

PAPPACHAN, P.; YUS, R.; MEHROTRA, S.; FREYTAG, J. Sieve: A middleware approach to scalable access control for database management systems. **Proc. VLDB Endow.**, v. 13, n. 11, p. 2424–2437, 2020.

PRACIANO, F. D. B. S.; AMORA, P. R. P.; ABREU, I. C.; MACHADO, J. C. Purpose Scan: A purpose-aware access method. **Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2022 Workshops, Poly and DMAH**, 2022.

REVIRIEGO, P.; APPLE, J.; LARRABEITI, D.; LIU, S.; LOMBARDI, F. On the privacy of adaptive cuckoo filters: Analysis and protection. **IEEE Trans. Inf. Forensics Secur.**, v. 19, p. 5867–5879, 2024.

REVIRIEGO, P.; SÁNCHEZ-MACIÁN, A.; GÓMEZ, E. M.; ROTTENSTREICH, O.; LIU, S.; LOMBARDI, F. Attacking the privacy of approximate membership check filters by positive concentration. **IEEE Trans. Computers**, v. 72, n. 5, p. 1409–1419, 2023.

REVIRIEGO, P.; SÁNCHEZ-MACIÁN, A.; WALZER, S.; GÓMEZ, E. M.; LIU, S.; LOMBARDI, F. On the privacy of counting bloom filters. **IEEE Trans. Dependable Secur. Comput.**, v. 20, n. 2, p. 1488–1499, 2023.

RIZVI, S.; MENDELZON, A. O.; SUDARSHAN, S.; ROY, P. Extending query rewriting techniques for fine-grained access control. **SIGMOD Conference**, ACM, France, p. 551–562, 2004.

SCHWARZKOPF, M.; KOHLER, E.; KAASHOEK, M. F.; MORRIS, R. T. Position: GDPR compliance by construction. **Poly/DMAH@VLDB**, Springer, California, p. 39–53, 2019.

SHASTRI, S.; BANAKAR, V.; WASSERMAN, M.; KUMAR, A.; CHIDAMBARAM, V. Understanding and benchmarking the impact of GDPR on database systems. **Proc. VLDB Endow.**, v. 13, n. 7, p. 1064–1077, 2020.

TRIPUNITARA, M. V.; CARBUNAR, B. Efficient access enforcement in distributed role-based access control (RBAC) deployments. **SACMAT**, ACM, p. 155–164, 2009.

XUE, Q.; ZHU, Y.; WANG, J.; LI, X. Distributed set intersection and union with local differential privacy. **ICPADS**, IEEE Computer Society, p. 198–205, 2017.