Query Answering in Multi-Relational Databases
Under Differential Privacy

by

Ios Kotsogiannis

Department of Computer Science
Duke University

Date: ______________________
Approved: ______________________

________________________________________
Ashwin Machanavajjhala, Supervisor

________________________________________
Jun Yang

________________________________________
Sudeepa Roy

________________________________________
Gerome Miklau

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2019
Abstract

Query Answering in Multi-Relational Databases Under Differential Privacy

by

Ios Kotsogiannis

Department of Computer Science
Duke University

Date: ____________________

Approved:

________________________
Ashwin Machanavajjhala, Supervisor

________________________
Jun Yang

________________________
Sudeepa Roy

________________________
Gerome Miklau

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2019
Abstract

Data collection has become a staple of both our digital and “off-line” activities. Government agencies, medical institutions, Internet companies, and academic institutions are among the main actors that collect and store users’ data. Analysis and sharing of this data is paramount in our increasingly data-driven world.

Data sharing provides a large positive societal value; however, it does not come cost-free: data sharing is at fundamental odds with individuals’ privacy. As a result, data privacy has become a major research area, with differential privacy emerging as the de facto data privacy framework. To mask the presence of any individual in the database, differentially private algorithms usually add noise to data releases. This noise is calibrated by the so called “privacy budget”, a parameter that quantifies the privacy loss allowed. One major shortcoming of both the definition and the supporting literature is that it applies to flat tables and extensions for multi-relational schemas are non trivial. More specifically, the privacy semantics in multi-relational schemas are not well defined since individuals might be affecting multiple relations each of which in a different degree. Moreover, there is no system that permits accurate differentially private answering of SQL queries while imposing a fixed privacy loss across all queries posed by the analyst.

In this thesis, we present PrivSQL, a first of its kind end-to-end differentially private relational database system, which allows analysts to query a standard relational database using a rich class of SQL queries. Our proposed system enables data
owners to flexibly specify the privacy semantics over the schema and provides a fixed privacy loss across all queries submitted by analysts. PrivSQL works by carefully selecting a set of views over the database schema, generating a set of private synopses over those views, and lastly answering incoming analyst queries based on the synopses. Additionally, PrivSQL employs a variety of novel techniques like view selection for differential privacy, policy-aware view rewriting, and view truncation. These techniques allow PrivSQL to offer automatic support for custom-tailored privacy semantics and permit low error in query answering.
The first principle is that you must not fool yourself and you are the easiest person to fool.

– Richard Feynmann
Contents

Abstract iv
List of Tables x
List of Figures xi
Acknowledgements xiii

1 Introduction 1
  1.1 Motivation & Problem Statement . . . . . . . . . . . . . . . . . . . 1
  1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  1.3 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2 Preliminaries & Notation 11
  2.1 Differential Privacy . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.2 Database & Queries . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Privacy for Relational Data 18
  3.1 The Case of Single Relation . . . . . . . . . . . . . . . . . . . . . . 18
  3.2 Defining Privacy for Multiple Relations . . . . . . . . . . . . . . . 19

4 Architecting a Differentially Private SQL Engine 23
  4.1 Goals & Design Principles . . . . . . . . . . . . . . . . . . . . . . . 23
  4.2 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 30
# Generating Private Synopses Based on Views

5.1 View Selection .................................................. 34
5.2 View Sensitivity Analysis ........................................ 38
5.3 Handling Complex Policies ...................................... 47
5.4 Private Synopsis Generator ................................... 54
5.5 Budget Allocator ................................................ 56
5.6 Privacy Proof .................................................. 58

# Optimizing Generation of a Single Synopsis

6.1 Background & Motivation ....................................... 60
6.2 Algorithm Selection ............................................ 66
6.3 Pythia Overview ................................................ 70
6.4 Delphi: Learning a FAS ....................................... 72
6.5 Deploying Pythia ............................................... 84

# Empirical Evaluation

7.1 PrivSQL Evaluation ............................................. 89
7.2 Pythia Evaluation .............................................. 102

# Related Work

8.1 Privacy Definitions ............................................ 115
8.2 Single Query Answering ....................................... 117
8.3 Multi-Query Answering ....................................... 118
8.4 Other .......................................................... 120

# Conclusions

9.1 Thesis Summary ............................................... 122
9.2 Future Directions ............................................. 123
List of Tables

5.1 Sensitivity estimation rules. ........................................... 39
5.2 Instantiations of BUDGETALLOC. ................................. 56
6.1 Algorithm overview for query release on single table. .......... 65
7.1 PrivSQL and input options used. .................................... 90
7.2 View Statistics for queries of $W_2$. ............................... 91
7.3 Pythia datasets overview. ........................................... 105
List of Figures

2.1 Queries supported by PrivSQL. ........................................... 15
3.1 Neighboring databases under foreign key constraints. ............... 21
4.1 Architecture of the PrivSQL System ................................. 30
5.1 An execution of VSELECTOR. ............................................. 35
5.2 Truncation rewrite operation (Alg 2) ................................. 41
5.3 Semijoin rewrite operation. ............................................. 53
6.1 The Pythia meta-algorithm. ............................................. 63
6.2 Delphi: Building of Pythia ............................................. 71
6.3 Example of an FAS for 2D range queries. ............................. 74
7.1 Relative error rates of PrivSQL. ....................................... 92
7.2 Relative error rates of PrivSQL for varying $\epsilon$. ............... 93
7.3 Comparison between PrivSQL and BASELINE_{Flex} ............. 96
7.4 Comparison of PrivSQL and FLEX in single query mode. .......... 98
7.5 Comparison of FLEX and PrivSQL sensitivity engines. ........... 98
7.6 Error rates of different configurations of PrivSQL. .............. 99
7.7 QERROR and bias rates for truncation-affected queries. .......... 100
7.8 Relative error and bias for truncation affected queries. .......... 101
7.9 Use Case: Workload Answering ....................................... 108 
7.10 Use Case: Naive Bayes Classifier (Gaussian) ..................... 111
7.11 Use Case: Naive Bayes Classifier (Multinomial) ................ 111
7.12 Criteria Comparison for Workload Answering .......................... 112
7.13 Tree learned by Delphi for the Short workload on 2D. ............... 112
7.14 Average Regret vs Shape .................................................. 113
7.15 Average Regret vs Domain Size ......................................... 114
7.16 Average Regret vs Scale ................................................... 114
Acknowledgements

It’s been a long journey, but not a lonely one. Looking back I’m the same, yet completely different. First and foremost, I would like to thank my advisor Ashwin; thank you for your patience and effort you put into making me a researcher. I also want to thank the members of the committee, my mentors, and all my collaborators throughout the years; thank you for making me a more well-rounded and better researcher. Of course, I want to thank the unsung heroes behind every successful graduate student, the staff in the Computer Science department: Marilyn, Melanie, Pam, Kathleen, Rita, Alison, Ashlee, and everybody I forget; you have all supported me way more than you know and you selflessly offered me not only assistance in student issues, but your friendship as well.

All my professors and teachers from back home, more notably: Sotiris Nikoletseas, Dimitris Kouzoudis, and Peter Triantafyllou at University of Patras, thank you for all your effort. Of course Siobhan Kidd without whom I would have needed another couple of years just to learn the language – and yes I stopped “doing” mistakes, I just make them now. My primary family back home Foivos, Ioli, Alexandra, Lina, and his majesty George the II; without you there is nothing. My secondary family not back in home: Kalari, Lauri, Alex, Ben, Jason, Animesh, Ali, Nisarg, Yuchao, Xi, Yianni & Nicolle, Ted & Giorgo, George, Jub & Anlo, and of course the Hyatt OGs; without you it’s boring.
1

Introduction

1.1 Motivation & Problem Statement

In our increasingly digital world, organizations like high tech companies, hospitals, and government agencies collect and store an abundance of user’s data. Analyses of this data provide immense business and societal value. More often than not and in order to perform such analyses, the data needs to be shared with a third party, whether that is an academic institution, a government agency, or even a contractor of the organization. Data sharing plays an integral role in our data-driven society as we highlight in the following examples.

For example, the U.S. Census Bureau performs the decennial census collecting information about people living in the United States. The decennial census consists of questionnaires filled by occupants in households in the United States. This data collection is highly crucial since policy decisions like congressional apportionment and redistricting are based directly on the Census data. Moreover, the Census Bureau also releases data products like the Summary File 1 (SF1)[Cen10] dataset which contain compiled data of the Census questions about all housing units, and the OnTheMap [MKA+08][Cen08] containing longitudinal employment data. Such data products are
an invaluable resource for scientists, policy makers, and local governments.

Internet companies are highly motivated to share their data with third parties, whether for business or research purposes. In the business front, a traditional route of revenue for social media companies is to share summaries of their users’ data to advertisers for more successful ad campaigns [Fac19]. However, sharing data is not always due to financial gains. For instance, Facebook recently announced a new initiative to allow social scientists to analyze their user data for research into the effect of social media on elections and more generally on democracy [KP18].

Medical institutions like clinics and hospitals collect patient data, which are often shared with medical researchers for new discoveries in their respective fields. Additionally, recent advancements have made possible the training of machine learning models for assisting physicians with patient diagnoses [Goo19]. These use cases highlight the importance of sharing patient data with a third party – the researchers in the former case and the physicians accessing the AI models in the latter case.

All previous examples emphasize that data sharing is inevitable as it promotes (a) immense economic growth, (b) wide expansion of scientific knowledge, (c) increased user experience. However, data sharing does not come for free as it is in fundamental conflict with user’s privacy. In contrast with data security, in the problem of data privacy the potential adversary is the very recipient of data sharing. These conflicting goals immensely complicate data sharing since it is not obvious how to even define what should be kept secret in the first place.

Privacy requirements of data sharing are often encoded in states’ legislation. For example, new privacy legislation in the EU and California heavily regulates the analysis and dissemination of user behavioral data, which includes all of their online activity. Similarly, the U.S. Census data releases are bound from Title 13, Chapter 9 of U.S. legislature, that states “Neither the secretary nor any officer or employee (...) make any publication whereby the data furnished by any particular establishment
or individual under this title can be identified (...)". In the case of sharing patient records in the United States, medical institutions and professionals are bound from the Health Insurance Portability and Accountability Act of 1996 (HIPAA).

In cases where privacy is not enforced – either from legislation or internal leadership – many problems arise. In a use case study in privacy leaks over micro-targeted advertisements in social networks [Kor10] the author could infer sensitive user information (like sexual orientation and religious preferences) just by creating ad campaigns on the Facebook ad platform. Facebook responded to that study with enforcing a threshold in the number of people targeted from each campaign. However, more recent work [ATV17, VAL +18, BHM +18] highlights that there is a systemic problem in how big organizations approach privacy issues in data sharing. More often than not, simple ad-hoc measures like data anonymization are inadequate to protect individuals' privacy.

These challenges have motivated and inspired a new line of work the past 20 years, with earlier proposed solutions [SS98, Swe02, MGKV06, LLV07] offering ad-hoc guarantees on the shared data. However, such approaches often lead to privacy leaks in presence of adversaries with sufficient background information, or with information of the technique used. A systemic approach to the problem of privacy should avoid the paradigm of “privacy by obscurity” – i.e., it should not rely on the adversary being agnostic of the privacy protocol utilized. Moreover, and due to the iterative process of data analysis, privacy engineers should be able to reason about the composition of multiple privacy algorithms operated on the sensitive data. This is something that the aforementioned work has failed to address.

Over the years the academic understanding of data privacy matured, resulting in the now widely accepted gold standard of private analysis: Differential Privacy (DP) [DR14]. An algorithm is differentially private if its output does not change significantly due to input changes. This ensures privacy when changes in the input
correspond to adding or removing an individual’s data, offering protections equivalent to plausible deniability. The privacy loss under differential privacy is quantified by a parameter \( \epsilon \), also called the privacy budget. Differential privacy is typically achieved by carefully injecting noise to true query answers, which results in a loss in the overall utility; with stronger privacy guarantees requiring an increasing amount of noise added. The privacy parameter \( \epsilon \) acts as a knob to this privacy/utility trade-off.

Recently, we have seen several real-world deployments of differential privacy in federal agencies like the US Census Bureau [MKA+08, HMA+17, Cen18] for publishing statistics; in companies like Uber [JNS18] for enabling a private query interface over user data for employees; and in Google[EPK14, BEM+17] and Apple [DPT17] applications for analyzing user data.

Despite the academic success and growing adoption, it is still extremely hard for non-experts to use differential privacy in practice. In fact, each of the deployments mentioned above has required a team of differential privacy experts to design algorithms and tune their parameters. In particular, it is difficult to both correctly define the privacy semantics as well as to design an algorithm which, given a fixed privacy budget and clear privacy semantics, offers the greatest accuracy for a task. Hence, each of the aforementioned deployments has required a team of privacy experts to design accurate algorithms that satisfy the privacy definition appropriate for the data. The challenges privacy experts need to address are multiple, starting even from answering simple queries on a single-relational schema and then moving to the more difficult problem of answering complex queries in a multi-relational schema. In the following, we present the most prominent challenges in the current landscape.

Complex Queries on Multi-Relational Schemas The algorithm design challenges are compounded when the input data are relational and have multiple tables. First, relational databases capture multiple entities and privacy can be defined at multiple
resolutions. For instance, in a relational schema involving persons and households, one could imagine two privacy policies – one hiding the presence of a single person record and another hiding the presence of a household record. The algorithms achieving the highest accuracy for each of the policies are different, and there is no known system that can automatically suggest an accurate differentially private mechanism given such privacy policies.

Second, there are no known algorithms for accurately answering complex queries over relational databases involving joins, groupby and correlated subqueries. Algorithms are known for accurately answering special classes of queries like statistical queries (e.g., histograms, CDFs, marginals on a single table) [Qar14, BBD+07, ZCX+14a, HLM12, XWG11, QYL13], sub-graph queries (e.g., triangle counting, degree distribution) [HLMJ09, KRSY11, KNRS13, DLL16, DZBJ18], and monotone queries (e.g., counts on joins) [CZ13]. A precursor to this work is PINQ [McS09a], a system that automatically adds the noise necessary for answering a limited set of SQL queries under $\epsilon$-differential privacy. The closest competitor to our work in terms of query expressivity is FLEX [JNS18], which only offers support for specific and limited privacy semantics that do not necessarily translate to real-world policies. FLEX does not support queries that have correlated subqueries or subqueries with groupby operations (e.g. it cannot support degree distribution queries).

Third, there are no known algorithms for accurately answering sets of complex queries under a common privacy budget. Sophisticated algorithms are known for optimally answering sets of statistical queries on a single table by identifying and adding noise to common sub-expressions [LHMW14]. Such mechanisms do not exist for graphs and SQL queries, and all prior work only optimizes error for single queries.

There is a growing line of work on privacy oriented programming frameworks [McS09b] and a few that focus on accuracy [ZMK+18] that lower the barrier to entry for non-experts to use DP. However, none of these frameworks has the capabilities of a
relational database. There is no support for declarative query answering; an analyst has to write a DP program themselves. Most systems only support queries on a single table and none consider updates to the database. While the need for such a system is obvious, building such a system requires solving several challenges, including defining privacy, accurately answering single and multiple queries under a privacy budget, as well as identifying a modular and extensible system architecture.

**Simple Queries on Single-Relational Schemas**  
Even the much simpler case of answering sets of linear counting queries on a single relation under the same privacy budget, turns out to be extremely non-trivial. In this case and for many data analysis tasks, the best accuracy achievable under $\epsilon$-differential privacy on a given input dataset is not known. There are general-purpose algorithms (e.g. the Laplace Mechanism [DMNS06] and the Exponential Mechanism [MT07]), which can be adapted to a wide range of settings to achieve differential privacy. However, the naive application of these mechanisms nearly always results in sub-optimal error rates. For this reason, the design of novel differentially-private mechanisms has been an active and vibrant area of research [HLM12][LHMW14][LYQ][QYL13]-[XGX12][ZCX+14a]. Recent innovations have had dramatic results: in many application areas, new mechanisms have been developed that reduce the error by an order of magnitude or more when compared with general-purpose mechanisms and with no sacrifice in privacy.

While these improvements in error are absolutely essential to the success of differential privacy in the real world, they have also added significant complexity to the state-of-the-art. First, there has been a proliferation of different algorithms for popular tasks. For example, in a recent survey [HMM+16], Hay et al. compared 16 different algorithms for the task of answering a set of 1- or 2-dimensional range queries. Even more important is the fact that many recent algorithms are data-dependent, meaning that the added noise (and therefore the resulting error rates)
vary between different input datasets. Of the 16 algorithms in the aforementioned study, 11 were data-dependent.

Data-dependent algorithms exploit properties of the input data to deliver lower error rates. As a side-effect, these algorithms do not have clear, analytically computable error rates (unlike simpler data-independent algorithms). When running data-dependent algorithms on a range of datasets, one may find that error is much lower for some datasets, but it could also be much higher than other methods on other datasets, possibly even worse than data-independent methods. The difference in error across different datasets may be large, and the “right” algorithm to use depends on a large number of factors: the number of records in the dataset, the setting of epsilon, the domain size, and various structural properties of the data itself.

Thesis Goal  The primary goal of this thesis is to lower the barrier to entry for non-experts by building a differentially private relational database that (a) supports privacy policies on realistic relational schemas with multiple tables, (b) allows analysts to declaratively query the database via aggregate queries involving standard SQL operators like JOINS, GROUPBY and correlated subqueries, (c) automatically designs a strategy with low error tuned to the privacy policy and analyst queries, and (d) ensures differential privacy with a fixed privacy budget over all queries posed to the system.

1.2 Contributions

The contributions of this thesis are the following:

- We propose a novel generalization of differential privacy in multi-relational databases with integrity constraints. More specifically, our generalization captures popular variants of differential privacy that apply to specialized examples of relational data (like Node- and Edge-DP for graphs). Moreover, it allows
the data owner to specify custom-tailored privacy semantics for the needs of his/her application.

- We design PrivSQL, a first of its kind end-to-end differentially private relational database system. PrivSQL permits data owners to specify privacy policies over a relational schema and exposes a differentially private SQL query answering interface to analysts. Moreover, the unique and modular architecture of PrivSQL allow for future extensions and improvements as new research innovations are proposed.

- PrivSQL employs a new methodology for answering complex SQL counting queries under a fixed privacy budget. Our algorithm identifies a set of views over base relations that support common analyst queries and then generates differentially private synopses from each view over the base schema. Queries posed to the database are rewritten as linear counting queries over a view and answered using only the private synopsis corresponding to that view, resulting in no additional privacy loss.

- PrivSQL utilizes a variety of novel techniques like policy-aware rewriting, truncation, and constraint-oblivious sensitivity analysis, to ensure that the private synopses generated from views provably ensure privacy as per the data owner’s privacy policy, and have high accuracy.

- We examine and formalize the problem of Algorithm Selection for answering simple queries on a single view of the data. More specifically, we define Algorithm Selection as the problem of choosing an algorithm from a suite of differentially private algorithms $\mathcal{A}$ with the least error for performing a task on a given input dataset. We require solutions to be (a) differentially private, (b) algorithm agnostic (i.e., treat each algorithm like a black box), and (c) offer
competitive error on a wide range of inputs. An algorithm’s competitiveness on a given input is measured using regret, or the ratio of its error to the minimum achievable error using any algorithm from $\mathcal{A}$.

- We present Pythia, a meta-algorithm for the problem of Algorithm Selection. Pythia uses decision trees over features privately extracted from the sensitive data, the workload of queries, and the privacy budget $\epsilon$. We propose a regret based learning method to learn a decision tree that models the association between the input parameters and the optimal algorithm for that input.

- We comprehensively evaluate PRIVSQL on both a use case inspired by the U.S. Census data releases and on the TPC-H benchmark. On a workload of $>3,600$ real world SQL counting queries and $\epsilon = 1$, 50% of our queries incurred $< 6\%$ relative error. In comparison, a system that uses the state-of-the-art FLEX[JNS18] incurs $> 100\%$ error for over 65\% of the queries; i.e., FLEX has worse error for these queries than a trivial baseline method that returns 0 for every answer (see Fig. 7.3b).

- We evaluate the performance of Pythia, our synopsis generator optimization tool on a total of 6,294 different inputs across multiple tasks and use cases (answering a workload of queries and building a Naive Bayes Classifier from sensitive data). On average, Pythia has low regret ranging between 1.27 and 2.27 (an optimal algorithm has regret 1).

1.3 Organization

The organization of this thesis is as follows. In Chapter 2 we define our notation and in Chapter 3 we present the privacy models for relational databases. In Chapter 4 we overview the architecture of PRIVSQL. Chapter 5 goes in depth of
how PrivSQL generates a set of private synopses over a multi-relational database. Chapter 6 presents Pythia, an optimization algorithm for generating a single private synopsis over a single view. In Chapter 7 we present our empirical evaluation. Chapter 8 offers an overview of prior related work. Lastly, in Chapter 9 we discuss limitations of PrivSQL and the future research directions.

Reading this thesis in the full sequential order is generally recommended for readers of all levels. However, alternative readings are also provided. Readers of high expertise in privacy literature, are recommended the following roadmap: 1 → 4 → 7 → 8, which skips technical details. Readers who want to learn more about the crucial details of PrivSQL and its privacy semantics should follow: 1 → 3 → 4 → 5 → 7 → 8. Readers interested in the simpler problem of answering linear counting queries on a single relation under differential privacy can read Chapter 6 in isolation.

The work in this thesis has also appeared in past publications, PrivSQL is presented first in [KTM+19] and [KTH+19], while Pythia was presented in [KMHM17], a demonstration of Pythia was also presented in [KHM+17].
2
Preliminaries & Notation

2.1 Differential Privacy

We first formally define our preferred privacy notion, differential privacy. Before doing so we need to introduce the notion of a database and neighboring databases.

The database \( D \) is a multiset of tuple and \( \mathcal{D} \) is the universe of valid databases. For a database \( D \) let \( N(D) \) be the neighborhood of \( D \), i.e., the set of all valid databases that differ from \( D \) by one tuple. More specifically,

\[
N(D) = \{ D' \mid D' \in \mathcal{D} s.t., |(D - D') \cup (D' - D)| = 1 \}
\]

The formal definition of differential privacy is then

**Definition 2.1.1 (Differential Privacy).** [DR14] A mechanism \( M : \mathcal{D} \to \Omega \) is \( \epsilon \)-differentially private if for any \( D \in \mathcal{D} \) and \( D' \in N(D) \) and \( \forall O \subseteq \Omega \):

\[
\frac{\Pr[M(D) \in O]}{\Pr[M(D') \in O]} \leq e^\epsilon
\]

Informally, the above definition implies that small changes in the input database do not significantly alter the output of the differentially private mechanism. This
provides indistinguishability between records in a database since data releases under differential privacy do not increase or decrease the posterior belief of an adversary about the presence or absence of a specific record. The parameter $\epsilon$ controls how much the output is allowed to differ for neighboring databases and is also referred as the privacy loss.

Differential privacy enjoys sequential and parallel composition which allow the privacy guarantee to gracefully degrade. More specifically:

**Theorem 2.1.1** (Sequential Composition [DR14]). Let $A_1, \ldots, A_k$ be differentially private algorithms, each satisfying $\epsilon_i$-differential privacy. Then their sequential execution on the same database $D$ satisfies $\sum \epsilon_i$-differential privacy.

**Theorem 2.1.2** (Parallel Composition [McS09a]). Let $A_1, \ldots, A_k$ be differentially private algorithms, each satisfying $\epsilon_i$-differential privacy. Let $D$ a database with a partition $\{D_1, \ldots, D_k\}$, where each partition is disjoint, i.e., $\forall i,j \in [k], i \neq j D_i \cup D_j = \emptyset$. Then the parallel execution $\{A_i(D_i)\}_{i \in [k]}$ satisfies $\max_i \epsilon_i - DP$.

The two composition theorems are invaluable tools that allow data owners to reason about the overall privacy loss on their data due to differentially private releases. Moreover, composition enables more complex algorithm design for better error guarantees. Lastly, note that the privacy loss parameter under the composition theorems can be thought of as a finite resource spent in different steps of a complex release. For that reason, $\epsilon$ is also referred to as the privacy loss budget or simply privacy budget.

The last property of differential privacy we present is robustness to post-processing. For an $\epsilon$-DP algorithm $A$, the privacy loss $\epsilon$ does not change under arbitrary post-processing of the output of $A$, as long as this post-processing does not access the sensitive data.
**Theorem 2.1.3** (Post-processing [DR14]). Let \( A : \mathcal{D} \rightarrow R \) an \( \epsilon \)-DP algorithm and any function \( f : R \rightarrow R' \). Then the composition of \( f \circ A : \mathcal{D} \rightarrow R' \) satisfies \( \epsilon \)-DP.

The design of differentially private algorithms is centered around the notion of function *sensitivity*. Much like stability properties, sensitivity measures how much the output of a function changes for “small” changes in the input database. Small changes in this context are captured from the notion of neighboring databases. More specifically:

**Definition 2.1.2** (Sensitivity). For a function \( f : \mathcal{D} \rightarrow \mathbb{R}^d \), let \( \Delta(f) \) its sensitivity:

\[
\Delta(f) = \max_{D \in \mathcal{D}, D' \in N(D)} \| f(D) - f(D') \|_1
\]

A basic differentially private algorithm for numerical queries, often used as a primitive block in more complex algorithms, is the *Laplace mechanism* [DR14]. The Laplace mechanism adds noise drawn from a Laplace distribution to the output of a numerical query. The distribution is parameterized based on the sensitivity of the query and the privacy parameter. More specifically:

**Definition 2.1.3** (Laplace mechanism). Given a function \( f : \mathcal{D} \rightarrow \mathbb{R}^d \) and a privacy parameter \( \epsilon \), the Laplace mechanism is defined as:

\[
M_{\text{lap}} = f(D) + \xi
\]

where \( \xi \) is a vector of \( d \) i.i.d. random variables drawn from \( \text{Lap}(0, \Delta(f)/\epsilon) \), i.e., the Laplace distribution with mean 0 and scale \( \Delta(f)/\epsilon \).

**Theorem 2.1.4** (Laplace mechanism). The Laplace mechanism as described in Definition 2.1.3 satisfies \( \epsilon \)-DP.

The Laplace mechanism exposes the relationship between the privacy parameter \( \epsilon \) and the necessary noise needed to provide the DP guarantee. High values of \( \epsilon \) require
less noise to satisfy at the cost of higher privacy loss and vice versa for small values of $\epsilon$. Thus, the privacy loss parameter $\epsilon$ can also be thought as a knob controlling the noise added in the data release.

2.2 Database & Queries

**Databases:** We consider databases with multiple relations $S = (R_1, \ldots, R_k)$, each relation $R_i$ has a set of attributes denoted by $\text{attr}(R_i)$. For attribute $A \in \text{attr}(R_i)$, we denote its full domain by $\text{dom}(A)$. Similarly, for a set of attributes $A \subseteq \text{attr}(R_i)$, we denote its full domain by $\text{dom}(A) = \prod_{A \in A} \text{dom}(A)$. An instance of a relation $R$, denoted by $D$, is a multi-set of values from $\text{dom}(%)$). We represent the domain of relation $R$ by $\text{dom}(R)$. For a record $r \in D$ and an attribute list $A \subseteq \text{attr}(R)$, we denote by $r[A]$ the value that an attribute list $A$ takes in row $r$.

**Frequencies:** For value $v \in \text{dom}(A)$, the frequency of $v$ in relation $R$ is the number of rows in $R$ that take the value $v$ for attribute list $A$; i.e., $f(v, A, R) = |\{r \in R | r[A] = v\}|$. We define the max-frequency of attribute list $A$ in relation $R$ as the maximum frequency of any single value in $\text{dom}(A)$; i.e., $\text{mf}(A, R) = \max_{v \in \text{dom}(A)} f(v, A, R)$. We will use max-frequencies of attributes to bound the sensitivity of queries.

**Foreign Keys:** We consider schemas with key constraints, denoted by $C$, in particular primary and foreign key constraints. A key is an attribute $A$ or a set of attributes $A$ that act as the primary key for a relation to uniquely identify its rows. We denote the set of keys in a relation $R$ by $\text{Keys}(R)$. A foreign key is a key used to link two relations.

**Definition 2.2.1.** Given relations $R, S$ and primary key $A_{pk}$ in $R$, a foreign key can be defined as:

$$S.A_{fk} \rightarrow R.A_{pk} \equiv S.A_{fk} \bowtie_{A_{pk}} R = S$$
<table>
<thead>
<tr>
<th>AggQuery</th>
<th>::= select count(*) from TableList</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableList</td>
<td>::= Table</td>
</tr>
<tr>
<td>Table</td>
<td>::= R</td>
</tr>
<tr>
<td>AttrList</td>
<td>::= A</td>
</tr>
<tr>
<td>Exp</td>
<td>::= Literal</td>
</tr>
<tr>
<td>Literal</td>
<td>::= A op A</td>
</tr>
<tr>
<td>op</td>
<td>::= =</td>
</tr>
</tbody>
</table>

**Figure 2.1:** Queries supported by PrivSQL. The terminal $R$ corresponds to one of the base relations in the schema, the terminal $A$ corresponds to an attribute in the schema and $val$ is a value in the domain of an attribute.

where the semijoin is the multiset $\{s \mid s \in S, \exists r, s[A] = r[B]\}$. That is, for every row in $s \in S$ there is exactly one row $r \in R$ such that $s[A_{fk}] = r[A_{pk}]$. We say that row $s \in S$ refers to row $r \in R$ ($s \rightarrow r$), and that relation $S$ refers to relation $R$ ($S \rightarrow R$).

The attribute (or set of attributes) $A_{fk}$ is called the foreign key.

We call a set of $k$ tables $D = (D_1, \ldots, D_k)$ a valid database instance of $(R_1, \ldots, R_k)$ under the schema $S$ and constraints $C$ if $D$ satisfies all the constraints in $C$. We denote all valid database instances under $(S, C)$ by $\text{dom}(S, C)$.

**SQL queries supported:** In Fig. 2.1 we present the grammar of PrivSQL supported queries. We consider aggregate SQL queries of the form $\text{SELECT COUNT(*) FROM } S \text{ WHERE } \Phi$, where $S$ is a set of relations and sub-queries, and $\Phi$ can be a positive boolean formula (conjunctions and disjunctions, but no negation) over predicates involving attributes in $S$. We support equijoins and subqueries in the WHERE clause, which can be correlated to attributes in the outer query. The grammar does not support negations, non-equi joins, and joins on derived attributes as
tracking sensitivity becomes a challenging and even intractable [AFG16] for such queries. PRIVSQL does not currently support other aggregations like sum/median but can be extended as discussed in Chapter 9.

2.2.1 Linear Queries

A subset of the supported grammar are linear counting queries on a single table – or linear queries for short. Answering linear queries under differential privacy is a well studied problem. We now introduce additional notation specific to linear queries on a single table.

A linear counting query on a single table, counts tuples on a table that satisfy a boolean formula on the attributes of that table.

**Definition 2.2.2 (Linear counting queries).** Using the grammar of Fig. 2.1, a linear counting query on a single table is defined as \( q ::= \text{SELECT COUNT}(*) \text{ FROM } R \text{ WHERE } \Phi \), where \( \Phi ::= A \text{ op } \text{val} | \Phi \text{ AND } \Phi | \Phi \text{ OR } \Phi \)

Similarly, a linear counting query on a single view over the base relations is defined with \( A \) being any attribute of the view.

A standard approach to answering linear queries on a single table under differential privacy is to use the vector representation of both the data and the queries. We introduce this notation here. We use bold, lowercase letters to denote column vectors, e.g. \( \mathbf{x} \). For a vector \( \mathbf{x} \) its \( i^{\text{th}} \) component is denoted with \( x_i \). We use bold uppercase letters to denote matrices, e.g. \( \mathbf{W} \). The transpose of a vector or a matrix are denoted with \( \mathbf{x}^\top \) and \( \mathbf{W}^\top \) respectively.

The representation of a single table \( R \) as a vector assumes that the attribute domain of \( R \) is discrete. Let \( \mathcal{A} = \{a_1, \ldots, a_d\} \) be the discrete domain of a relation \( R \) and \( D \) an instantiation of \( R \), then we can describe \( D \) as a vector \( \mathbf{x} \in \mathbb{N}^d \), where \( x_i \) counts the number of tuples in \( D \) with value \( a_i \).
Similarly, a linear counting query over a table $R$ can be expressed as a vector over the domain of $R$: $q \in [0, 1]^d$. Then, a workload of $m$ linear queries is an $m \times d$ matrix where each row represents a different linear query. For an instance $D$ with vector representation $x$ and a query workload $W$, the answer to this workload is defined as $y = Wx$. 
3

Privacy for Relational Data

3.1 The Case of Single Relation

The formal definition of differential privacy (DP) considers a database consisting of a single relation:

**Definition 3.1.1** (DP for Single Relation). A mechanism $\mathcal{M} : \text{dom}(R) \rightarrow \Omega$ is $\epsilon$-differentially private if for any relational database instance $D \in \text{dom}(R)$ of size at least 1 and $D' = D - \{t\}$, and $\forall O \subseteq \Omega$:

$$|\ln(\Pr[\mathcal{M}(D) \in O]/\Pr[\mathcal{M}(D') \in O])| \leq \epsilon$$

The above definition implies that deleting a row from any database does not significantly increase or decrease the probability that the output of the mechanism lies in a specific set. Note that this is equivalent to the standard definition of differential privacy Definition 2.1.1 that requires the output of the mechanism be insensitive to deleting or adding a row in $D$.

However, defining privacy for a schema with multiple relations is more subtle. First, we need to determine which relation(s) in the schema is(are) private. Second,
adding or removing a record in a relation can cause the addition and/or removal of multiple rows in other relations due to schema constraints (like foreign key relationships).

3.2 Defining Privacy for Multiple Relations

Given a database relational schema $S$, we define a privacy policy as a pair $P = (R, \epsilon)$, where $R$ is a relation of $S$ and $\epsilon$ is the privacy loss associated with the entity in $R$. We refer to relation $R$ as the primary private relation. The output of a mechanism enforcing $P = (R, \epsilon)$ does not significantly change with the addition/removal of rows in $R$.

To capture privacy policies and key constraints, we propose a definition of neighboring tables inspired by Blowfish privacy [HMD14]. For two database instances $D$ and $D'$, we say that $D$ is a strict superset of $D'$ (denoted by $D \supset D'$) if (a) $\forall i, D_i \supseteq D_i'$ and (b) $\exists i, D_i \supset D_i'$. That is, all records that appear in $D'$ also appear in $D$ and there is at least one row in a relation of $D$ that does not appear in $D'$.

**Definition 3.2.1 (Neighboring Databases).** Given a schema $S$ with a set of foreign key constraints $C$, and a privacy policy $P = (R_i, \epsilon)$, for a valid database instance $D = (D_1, \ldots, D_k) \in \text{dom}(S, C)$, we denote by $\ominus_C(D, R_i)$ a set of databases such that $\forall D' \in \ominus_C(D, R_i)$:

- $\exists r \in D_i$, but $r \notin D_i'$, and
- $D'$ satisfies $C$, and
- $\not\exists D''$ that satisfies $C$ and $D \supset D'' \supset D'$.

That is, $D'$ is a valid database instance that results from deleting a minimal set of records from $D$, including $r$. We call database instances $D, D'$ neighboring databases w.r.t. relation $R_i$ if $D' \in \ominus_C(D, R_i)$. 

19
Example 1. Consider the database of Fig. 3.1a with schema **Person** (pid, age, hid) and **Household** (hid, st, type). **Person**.hid is a foreign key to **Household**. Fig. 3.1b shows a neighboring instance of the original database under privacy policy $P = (\text{Person}, \epsilon)$. Notice that in that instance, the **Household** table is unchanged and only person $p\text{10}$ is removed. However, under the privacy policy $P = (\text{Household}, \epsilon)$ (Fig. 3.1c) removing $h\text{02}$ from **Household** results in deleting two rows in **Person** table. In this case, neighboring databases differ in both the primary private relation **Household** as well as a secondary private relation **Person**.

**Definition 3.2.2** (Secondary Private Relations). Let $\mathbb{S}$ be a schema with constraints $C$ and $P = (R_i, \epsilon)$ be a privacy policy. Then a relation $R_j \in \mathbb{S}$ is a secondary private relation iff: $\exists D \in \text{dom}(\mathbb{S}, C), \exists D' \in \ominus_C(D, R_i)$ s.t. $D_j \neq D'_j$.

We call a policy that results in no secondary private relations (e.g., **Person** policy in Fig. 3.1b) a *simple* policy. In this case, neighboring tables differ in only the primary private relation in exactly one row. We call policies that result in secondary private relations (e.g. **Household** policy in Fig. 3.1c) as *complex* policies.

**Definition 3.2.3** (DP for Multiple Relations). Given a schema $\mathbb{S}$ with foreign key constraints $C$ and privacy policy $P = (R, \epsilon)$ be a policy. A mechanism $\mathcal{M} : \text{dom}(\mathbb{S}, C) \rightarrow \Omega$ is $P$-differentially private if for every set of outputs $O \subseteq \Omega$, $\forall D \in \text{dom}(\mathbb{S}, C)$, and $\forall D' \in \ominus_C(D, R)$:

$$|\ln(Pr[\mathcal{M}(D) \in O]/Pr[\mathcal{M}(D') \in O])| \leq \epsilon$$

As in standard differential privacy, our definition permits sequential composition:

**Theorem 3.2.1** (Sequential Composition). Given a schema $\mathbb{S}$ with constraints $C$, let mechanisms $M_1, M_2$ that satisfy $P_1$-DP and $P_2$-DP, with $P_i = (R, \epsilon_i)$. Then the sequence of $M_1$ and $M_2$ satisfies $P_{\text{seq}}$-DP, with $P_{\text{seq}} = (R, \epsilon_1 + \epsilon_2)$. 
Global Sensitivity: Designing differentially private mechanisms requires an important notion called *global sensitivity* – the maximum change to the query output in neighboring datasets. In multi-relational databases, the sensitivity of a query can change depending on which relation is identified as the primary private relation. We denote by \( \Delta_R \) the sensitivity of a query with respect to relation \( R \in \mathcal{S} \).

A query that outputs another relation is called a *view*. A change in a view is measured using symmetric difference, and the global sensitivity of a view is defined as follows:

**Definition 3.2.4** (Global Sensitivity for View). Given a schema \( \mathcal{S} \) with foreign key constraints \( \mathcal{C} \) and privacy policy \( P = (R, \epsilon) \). A view query \( V \) takes as input an instance of the database \( D \) and outputs a single relation instance \( V(D) \). The global sensitivity of \( V \) w.r.t. \( R \) is defined as the maximum number of rows that change in \( V \) across neighboring databases w.r.t. \( R \), i.e.,

\[
\Delta^C_R(V) = \max_{D \in \text{dom}(\mathcal{S}, \mathcal{Q})} \Delta^C_R(V, D) \quad (3.1)
\]

\[
\text{where,} \quad \Delta^C_R(V, D) = \max_{D' \in \mathcal{C}(D, R)} V(D) \Delta V(D') \quad (3.2)
\]

is the down sensitivity of a given instance \( D \) and \( A \Delta B = (A \setminus B) \cup (B \setminus A) \) denotes symmetric difference.
**Composition:** \((R, \epsilon)-differential privacy satisfies composition rules like regular differential privacy.

**Theorem 3.2.2** (Sequential Composition). Given multiple relations \((R_1, \ldots, R_k)\) with foreign key constraints specified in schema \(S\). Let \(R_i\) be the primary private relation. The sequential execution of mechanisms \(M_1, \ldots, M_k\), where \(M_j\) satisfies \((R_i, \epsilon_j)-DP\) on a database instance \(D \in dom_S(R_1, \ldots, R_k)\) is also \((R_i, \epsilon)-differentially private with parameter \(\epsilon = \sum_{j=1,\ldots,k} \epsilon_j\).

**Relationship to Other Privacy Notions:** Most variants of differential privacy that apply to relational data can be captured using a single private relation and foreign key constraints on an acyclic schema [AFG16, CZ13, KRSY11, KNRS13, DNPR10, LMG14]. For instance, a graph \(G = (V, E)\) can be represented as a schema with relations \(\text{Node}(id)\) and \(\text{Edge}(src\_id, dest\_id)\) with foreign key references from \(\text{Edge}\) to \(\text{Node}\) \((src\_id \rightarrow id\) and \(dest\_id \rightarrow id\)). Edge-DP [KRSY11] is captured by \(P\)-DP by setting \(\text{Edge}\) as the primary private relation \(R\), Node-DP [KNRS13] is captured if we set \(\text{Node}\) as \(R\). Under the latter policy, neighboring databases differ in one row from \(\text{Node}\) and all rows in \(\text{Edge}\) that refer to the deleted \(\text{Node}\) rows. Similarly, user-level- and event-level-DP are also captured using a database schema \(\text{User}(id, \ldots), \text{Event}(eid, uid, \ldots)\) with events referring to users via a foreign key \((uid \rightarrow id)\). By setting the \(\text{Event} (\text{User})\) as the primary private relation, we get Event-DP (User-DP, resp.) [DNPR10].

The privacy model in FLEX [JNS18] considers neighboring tables that differ in exactly one row in one relation. FLEX does not capture standard variants of DP described above since the FLEX privacy model ignores all constraints in the schema. For instance, using FLEX for graphs would consider neighboring databases that differ in exactly one edge or one node, but never in all the edges connected to a node. Thus, FLEX’s privacy model cannot capture Node-DP.
Architecting a Differentially Private SQL Engine

4.1 Goals & Design Principles

PRIVSQL is designed to meet three central goals:

- **Bounded Privacy Loss**: The system should answer a workload of queries with bounded privacy loss.

- **Support for Complex Queries**: Each query in the workload can be a complex SQL expression over multiple relations.

- **Multi-resolution Privacy**: The system should allow the data owner to specify which entities in the database require protection.

While there is prior work that addresses each of these in isolation, there is no prior work, to our knowledge, that supports two or more goals simultaneously. For instance, in [JNS18] the authors propose differentially private techniques for answering a single (SQL) query given a fixed privacy loss budget. Such an approach does not extend naturally to answering a workload of queries as the privacy loss compounds for each new query that is answered. Further, the “fundamental law of
information reconstruction” [DN03] suggests that running such a system indefinitely would leak enough information to rebuild the entire database – or the system must inject increasingly larger amounts of noise into query answers.

In the rest of this chapter, we outline the key design principles that enable PRIVSQL to support these goals and then describe the system architecture.

**Principle 1.** *Differentially private queries should not be answered on the live database. Rather, queries should be answered on a privately-constructed synopsis of the database.*

Prior work (e.g. FLEX) has proposed privately answering SQL queries by (a) querying the live database and (b) adding noise calibrated to the sensitivity of the query. In contrast, we argue that a differentially private query answering system must be divorced from a live database which may undergo continuous updates. Such a decoupling allows for a constant privacy loss, secures from side channel attacks, and lastly, offers consistency across queries for free. We explain each of these below:

**Constant Privacy Loss**  All interactions between the database and the analyst must be differentially private – i.e., no matter how many queries an analyst poses, her view of the database, and the process that constructs it, all interactions must satisfy $\epsilon$-differential privacy, where $\epsilon$ is a pre-specified privacy budget. If the system answered queries on the live database, then each query would use up a part of the privacy budget and the system would have to shut down after relatively few queries. For instance, in FLEX, if each query is answered under 0.1-DP, then a total budget of 1.0 only allows up to 10 queries.

To support a workload of queries, our first key idea is to construct *synopses*. A synopsis captures important statistical information about the database that is useful for answering many queries (analogous to pre-computed samples in approximate query processing ss [AMP+13]). The privacy loss budget is spent constructing and
releasing the synopses. Once released, subsequent queries are answered using only the synopsis and not the private database. Since the synopsis is public, there is no privacy cost to querying it and an unlimited number of queries can be answered – though the fundamental law also implies that some query answers will be poorly approximated, see Principle 2 for further discussion.

Side Channel Attacks Answering queries on a live database has safety issues – the observed execution time to answer a query on the live database could break the differential privacy guarantee and reveal sensitive properties about the records in the database. For instance, consider a table storing properties of nodes (in a node table) and edges (in an edge table) in a social network. Suppose the analyst queries for the number of edges connected to users over the age of 90. Suppose Bob is the only person in the database with age > 90 and has a thousand friends. With Bob in the database, the query answer would be 1000. If Bob’s record were not in the database, the answer to the query is 0. Any differential privacy mechanism for answering this query would add enough noise to obfuscate this difference. However, a typical DP mechanism (like FLEX) would not hide the time taken to compute the answer. Without Bob, the live database would identify this query as joining an empty intermediate table with the edge table, and hence would return quickly. On the other hand, with Bob in the database, the join may take perceptibly more time, thus revealing the presence of Bob.

Such timing attacks are avoided if analysts are only exposed to a private synopsis over the data that is constructed offline. Continuing the above example, the private synopsis generation may take more or less time depending on whether Bob’s record is in the database, but this is hidden from the analyst who only interacts with the private synopsis.
**Consistency**

Typical differentially private mechanisms work by adding random noise to query answers. Therefore, if queries were answered on the live database, an analyst would see different query answers to the same queries – unless the system cached previous queries and answers; which is indeed akin to maintaining a synthetic database. Moreover, relationships between queries may also be distorted. For instance, due to noise, the total number of males in a dataset could be smaller than the number of males of age 20-50 (while in the true data the reverse must clearly be true). If one were answering queries on the live database (like in FLEX), the burden of making noisy answers consistent would be shifted to the analyst.

Since we propose to generate a private synopsis, which is already differentially private, (a) no further noise needs to be added and (b) we can ensure that the private synopsis is consistent. A downside of answering queries on a private synopsis is that updates to the database are not reflected in the query answers. We discuss this in more detail in Chapter 9.

**Principle 2.** The private synopsis must be tuned to answer queries for an input query workload.

**Synopses generated for selected views**

There is considerable prior work on generating a differentially private statistical summary for a single table. Such strategies have been shown to support workloads of simple (linear) queries. But if a synopsis were generated for each base table in the schema, it is known that complex queries, such as the join of two tables, would be poorly approximated [MPRV].

This motivates the second key idea: to support complex queries, we select a set of (complex) views over the base tables and then generate a synopsis for each of the selected views. Our approach is based on the assumed availability of a representative workload, a set of queries that captures, to a first approximation, the kinds of queries that users are likely to ask in the future. Views are selected so that each query in
the representative workload can be answered with a linear query on a single view. Intuitively, views encode the join structures that are common in the workload.

The celebrated result by Dinur-Nissim [DN03], the Fundamental Law of Information Reconstruction, shows that a database containing \( n \) bits can be accurately reconstructed by an adversary that submits \( n \log^2 n \) counting queries, even if each of the queries has \( o(\sqrt{n}) \) additive noise. This implies that we cannot hope to accurately answer too large a set of queries from any single synopsis under strong privacy guarantees. It therefore means that we must specify as input a representative workload of queries to be answered. This workload can be either a list of explicitly defined queries, or a set of parameterized queries – where constants are replaced by wildcards. The private synopsis will be designed to provide answers to the representative workload with high accuracy. Of course, if the workload contains too many queries then we can not answer all of them with high accuracy without violating the Fundamental Law of Reconstruction. Thus our accuracy guarantees on the queries in the representative workload are best-effort. Our system also tries to answer queries that are not in the input workload and if it can’t, then it informs the user.

**Principle 3.** Private synopses may need to be generated over views defined on the base tables and not just on the base tables.

Prior work has shown that queries involving the join of two tables cannot be answered accurately just using private synopses that have been generated independently from each of the tables. For instance, Mironov et al. [MPRV] show a \( \Omega(\sqrt{n}) \) lower bound on the error of computing the intersection between two tables given differentially private access to the individual tables (and not their join). The intuition behind this result follows from the definition of differential privacy. Since join keys are typically unique, no differentially private algorithm can preserve the key. Thus, joins have to be done on coarser quasi-identifiers which are associated with a
sufficiently large number of tuples.

In contrast, given access to a view that encodes the join over the two base tables, computing the size of the join is a counting query that can be answered with constant error. Thus, if one expects to receive many queries involving the join between two tables, the system must generate private synopses from an appropriate view over the base tables and not just from the base tables themselves.

**Principle 4. View sensitivity must be bounded and tractable.**

*View sensitivity bounded using rules and truncation:* When PrivSQL generates a synopsis for each view, it ensures the synopsis generator is differentially private with respect to its input, a view instance. A subtle but important point is that achieving \( \epsilon \)-differential privacy with respect to a view does not imply \( \epsilon \)-differential privacy with respect to the base relations from which the view is derived. This is because a single change in a base relation could affect multiple records in the view. For example, imagine a view that describes individuals living in households along with employment characteristics of the head of household. Changing the employment status of the head of an arbitrary household would affect the records of all members of that household. To correctly apply differential privacy, we must know (or bound) the view sensitivity, which is informally defined as the worst-case change in the view due to the insertion/deletion of a single tuple in a base relation.

This brings us to the third key idea: we introduce novel techniques for calculating a bound on view sensitivity. Exact sensitivity calculation is hard, even undecidable [AFG16]. We employ a rule-based calculator to each relational operator in the view definition (which is expressed as a relational algebra expression). The per operator bounds compose into an upper bound on the global sensitivity of the view.

An additional challenge is that some queries have high, even unbounded, sensitivity because of worst case inputs. The previous example has a sensitivity that is
equal to the size of the largest possible household. Our approach to addressing high
sensitivity queries is to use truncation to drop records that cause high sensitivity
(e.g., large households). By lowering sensitivity, truncation lowers the variance in
query answers at the expense of introducing bias that arises from data deletion. We
describe techniques for using the data to privately estimate the truncation threshold
and we empirically explore the bias-variance trade-off.

**Principle 5.** *Sensitivity estimation should be policy agnostic.*

*Privacy at multiple resolutions:* A key design goal of PrivSQL is to allow data owners
to select the privacy policy that is most appropriate to their particular context.
Differential privacy, as formally defined, assumes the private data is encapsulated
within a single relation. Adapting it to multi-relational data is non-trivial, especially
given integrity constraints like foreign key constraints. When a tuple is removed from
one relation, it can cause (cascading) deletions in other relations that are linked to
it through foreign keys.

Our fourth key idea is extending differential privacy to the multi-relational set-
ting. With our approach, one relation is designated as the *primary private relation*,
but the privacy protection extends to other *secondary private* relations that refer to
the primary one through foreign keys. We show this allows the data owner to vary
the privacy resolution (e.g., to choose between protecting an individual vs. an entire
household and all its members). We describe this extension in Section 3.2 and relate
it to prior literature.

*View rewriting allows policy flexibility:* The challenge with supporting flexible privacy
policies is that now view sensitivity will depend on the policy. For example, a policy
that protects entire households would generally have higher sensitivity than a policy
that protects individuals. PrivSQL is designed to offer the data owner flexibility
to choose the appropriate policy and the system will automatically calculate the appropriate sensitivity.

The fifth and final key idea is that we use view rewriting to ensure correct, policy-specific sensitivity bounds. Rewriting makes explicit whether a view depends on the primary private relation, even in cases when the view does not mention it! After rewriting, downstream components (such as sensitivity calculation and synopsis generation) can be oblivious to the particular policy and apply conventional differential privacy on the primary private relation.

4.2 System Architecture

We now review the architecture of PRIVSQL (illustrated in Fig. 4.1) and the algorithms of the two main operational phases. The first phase is the synopsis generation phase where a representative workload is used to guide the selection of views followed by the differentially private construction and publication of a synopsis for each of the selected views. Next is query answering phase where each user query is mapped to the appropriate view and then answered using the released synopsis of that view.

Synopsis generation phase As described in Algorithm 1, this phase takes as input a database instance $D$, which is private, and its schema $S$, which is considered public. It also takes a representative query workload of SQL queries, $Q$, and a privacy policy
Algorithm 1 SYNOPSIS-GENERATION

Require: Schema $S$, database $D$, representative workload $Q$, privacy policy $P = (R, \epsilon)$.

Ensure: A set of views $V$ and private synopses $\{\tilde{S}_V\}_{V \in V}$

1: $V \leftarrow VSELECTOR(S, Q)$   \texttt{\> Choose views based on workload}$
2: \text{Reserve $\epsilon_{mf}$ to estimate thresholds for relations in views.}$
3: $\epsilon \leftarrow \epsilon - \epsilon_{mf}$
4: for each view $V$ in $V$ do
5: \hspace{1em} $V^{\tau, \ominus} \leftarrow VREWRITER(V, P, S)$
6: \hspace{1em} $\tau_V \leftarrow$ Estimate truncation thresholds using $\epsilon_{mf}/|V|$
7: \hspace{1em} $\hat{\Delta}_V \leftarrow$ SENS CALC($V^{\tau, \ominus}, S, \tau_V$)
8: \hspace{1em} $Q_V \leftarrow \{\bar{q} \mid q \in Q \land QTRANSFORM(q, S) = (\bar{q}, V)\}$
9: end for
10: for each $V \in V$ do
11: \hspace{1em} $\epsilon_V \leftarrow$ BUDGETALLOC($V, [Q_V]; [\hat{\Delta}_V], \epsilon$)
12: \hspace{1em} $\tilde{S}_V \leftarrow PRIVSYNGEN(V^{\tau, \ominus}, V^{\tau, \ominus}(D), \epsilon_V, Q_V$)
13: end for
14: return $(V, \tilde{S}_V)$ for each $V \in V$

Algorithm 2 QUERY-ANSWERING

Require: Query $q$, schema $S$, views $V$, synopses $\tilde{S}$.

Ensure: Query answer or $\bot$

1: $(\bar{q}, V) \leftarrow QTRANSFORM(q, S)$
2: if $V \in V$ then
3: \hspace{1em} return COMPUTEQUERYANSWER($\bar{q}, \tilde{S}_V$)
4: else
5: \hspace{1em} return $\bot$
6: end if

$P = (R, \epsilon)$ that specifies a privacy budget $\epsilon$ and a \textit{primary private relation} $R$ (formally defined in Section 3.2).

First, the $VSELECTOR$ module (line 1) uses the representative workload $Q$ to select a set of view definitions $V$.

Next, each view (interpreted as a relational algebra expression) is rewritten using the $VREWRITER$ module (line 5) in two ways. First, \textit{truncation} operators are included when there is a join on at attribute that may result in a potentially unbounded number of output tuples. The truncation operator enforces a bound on join size by throwing away join keys with a multiplicity greater than a threshold. The thresholds can be learnt from the data (line 6) in a differentially private manner. Next, base tables in the view definition are rewritten using \textit{semijoin expressions}, which makes explicit the foreign key dependencies between the primary private relation and other
base tables. This ensures that the computed sensitivity matches the privacy policy.

Next, the SensCalc module (line 7) computes for each rewritten view \( V \), an upper bound on the global (or worst case) sensitivity \( \hat{\Delta}_R(V) \). The sensitivity bound \( \hat{\Delta}_V \) is used in the privacy analysis and affects how much privacy loss budget is allocated to each view.

Synopsis generation for each view is guided by a partial workload \( Q_V \), which is the set of queries from the representative workload \( Q \) the can be answered by this view. The set \( Q_V \) is constructed (line 8) by applying the function QTransform (constructed by VSelector) to each query in \( Q \). This function transforms a query \( q \) into a pair \( (\bar{q}, V) \) where \( \bar{q} \) is a new query that is linear (or a simple aggregation without involving joins) on view \( V \).

Lastly, and for each view \( V \) we generate a private synopsis. Each synopsis is allocated a portion of the total privacy loss budget. The BudgetAlloc component (line 11) determines the allocation based on factors like view sensitivity and/or the size of \( Q_V \). Finally, the PrivSynGen component takes as input the view definition, view instance \( V(D) \), a set of linear queries \( Q_V \), and a privacy budget \( \epsilon_V \) and returns a differentially private private synopsis \( \tilde{S}_V \). This module runs an \( \epsilon_V \)-differentially private algorithm and outputs either a set of synthetic tuples or a set of query answers – like histograms or a set of counts.

We present our generalization of differential privacy for relational databases in Section 3.2. We outline VSelector in Section 5.1. We describe SensCalc and the truncation rewrite in Section 5.2, and the semijoin rewrites in Section 5.3. PrivSynGen and BudgetAlloc are described in Sections 5.4 and 5.5 respectively. Lastly, the privacy proof of PrivSQL is presented in Section 5.6.

**Query answering** using views is a well studied problem [Hal01] and in PrivSQL is performed by the query answering phase. More specifically, it uses the function
QTransform, described above, to convert $q$ into a query $\bar{q}$ that is linear on a view $V$. If $V$ is one of the views for which PrivSQL generated a synopsis, then $\bar{q}$ is then executed on the appropriate private synopsis to produce an answer. If the query cannot be mapped to any view, it returns $\bot$. As our techniques for query answering are straightforward, we omit further details.

End-to-End Privacy  Executing an $\epsilon_V$-DP algorithm on $V(D)$ can be shown to satisfy $\hat{\Delta}_V \epsilon_V$-DP over the base tables [McS09b].

The overall privacy of PrivSQL follows from the sequential composition property of differential privacy [DR14]. As long as the budget allocation satisfies:

$$\sum_{V \in V} \hat{\Delta}_V \epsilon_V \leq \epsilon - \epsilon_{mf}$$  \hspace{1cm} (4.1)

where $\epsilon_{mf}$ is the budget allocated to learning truncation thresholds, then, PrivSQL always satisfies the policy-specific privacy guarantee with privacy loss of $\epsilon$ (see Section 5.6). Note that query answering has no privacy cost.
Generating Private Synopses Based on Views

5.1 View Selection

View selection in PrivSQL is performed by the VSELECTOR module, which takes as input a set of representative queries \( Q \) over the schema \( S \) and returns \( (\mathcal{V}, \text{QTRANSFORM}) \). \( \mathcal{V} \) is a set of views such that all queries of \( Q \) are linearly answerable using some view \( V \in \mathcal{V} \). QTRANSFORM is an internal function of VSELECTOR that transforms queries of \( Q \) and helps generate the set of views \( \mathcal{V} \). Our system exposes QTRANSFORM outside VSELECTOR so that other components of PrivSQL can map new queries to the set of views \( \mathcal{V} \).

**Definition 5.1.1.** A query \( q \) over schema \( S \) is answerable using a view \( V \) if there is a query \( \bar{q} \) defined on the attributes in \( V \) such that for all database instances \( D \in \text{dom}(S) \), we have, \( q(D) = \bar{q}(V(D)) \). Additionally, we say that \( q \) is linearly answerable using \( V \), if \( \bar{q} \) is linear on \( V \).

Linear answerability ensures that queries in \( Q \) can be directly answered from some \( V \in \mathcal{V} \) without additional join or group-by operations. Moreover, the privacy analysis of sets of linear queries is easy and it allows the use of well known workload-
In Fig. 5.1 we show an execution of VSelector on workload $Q = \{q_1, q_2, q_3, q_4\}$, for which VSelector produces two distinct views $V_1$ and $V_2$, under which all queries of $Q$ are linearly answerable. More specifically, $q_1$ and $q_2$ can be answered using linear queries $\bar{q}_1$ and $\bar{q}_2$ on $V_1$. Similarly, $q_3$ and $q_4$ can be answered using linear queries $\bar{q}_3$ and $\bar{q}_4$ on $V_2$. For the remainder we denote the transformed workloads $Q_{V_1} = \{\bar{q}_1, \bar{q}_2\}$ and $Q_{V_2} = \{\bar{q}_3, \bar{q}_4\}$ as the partial workloads of views $V_1$ and $V_2$ respectively.

### 5.1.1 Design Considerations:

The goal of VSelector is to produce views such that (a) all queries of $Q$ can be answered from a view and (b) the total privacy loss of PrivSQL as expressed in Eq. (5.8) is minimized.

An initial approach to minimize the privacy loss is to release a single view $V_{\text{one}}$. Let VSelector$_{\text{one}}$ denote this approach, with $V_{\text{one}}$ the universal view constructed
by joining all relations under key-foreign key constraints. It is clear that under $V_{one}$ all queries of $Q$ are answerable. However, $V_{SELECTOR_{one}}$ does not guarantee linear answerability – see $q_3$ and $q_4$ of Fig. 5.1 that are not linearly answerable using $V_{one}$, as they require self joins on the Person relation. In addition, $V_{SELECTOR_{one}}$ does not necessarily minimize the privacy loss of Eq. (5.8) since the factor $\hat{\Delta}_{V_{one}}$ will be as large as the largest sensitivity of a query answered from $V_{one}$. This penalizes low sensitivity queries, as they will be answered by the high sensitivity view $V_{one}$.

Another way to minimize the privacy loss is to generate views with a small $\Delta_V$ value. This can be achieved from $V_{SELECTOR_{all}}$, that for each query $q \in Q$ returns a view $V_q$ containing all tuples that $q$ accesses. Evidently, $V_{SELECTOR_{all}}$ satisfies linear answerability for all queries of $Q$, since a query $q$ is linearly answerable by the simple linear query $\tilde{q} = \text{SELECT COUNT(*) FROM } v_q$. Moreover, all views $V_q$ returned from $V_{SELECTOR_{all}}$ have the smallest possible $\Delta_{V_q}$. Still, $V_{SELECTOR_{all}}$ does not minimize the privacy loss, as it fails to take advantage of parallel composition [DR14] between queries of $Q$. For instance, consider queries $q_1$ and $q_2$ from Fig. 5.1 that have no overlap – as $q_1$ counts underage people, and $q_2$ counts heads of households over 21 years old. For these queries, $V_{SELECTOR_{all}}$ will create views $V_1$ and $V_2$, resulting in synopses $\tilde{S}_{V_1}$ and $\tilde{S}_{V_2}$ generated with privacy budgets $\epsilon_{V_1}$ and $\epsilon_{V_2}$ s.t. $\epsilon = \epsilon_{V_1} + \epsilon_{V_2}$. However, both queries could be answered from a single synopsis $\tilde{S}_V$ generated with a total privacy budget of $\epsilon$, resulting in higher accuracy answers.

5.1.2 Approach

We propose a heuristic algorithm $V_{SELECTOR}$ that: (a) satisfies linear answerability w.r.t. $Q$, (b) each partial workload $Q_V$ contains a non-trivial number of queries for efficient query sensitivity analysis, (c) each $Q_V$ is sensitivity homogeneous, and (d) returned views have low complexity for tractable sensitivity analysis.

---

1 If the schema is not semijoin-reduced, then joining all relations using the foreign keys does not capture all rows of all base tables. We ignore this detail since we do not use the universal relation approach to view selection.
Algorithm 1 QTransform $(q, S)$

- $V, ar{q} \leftarrow f_b(q, S)$  \hfill $\triangleright$ Baseline transformation
- $V \leftarrow f_{dc}(V)$  \hfill $\triangleright$ Decorrelate predicates of $V$
- $V, ar{q} \leftarrow f_{pt}(V, ar{q})$  \hfill $\triangleright$ Transfer non-join predicates to $\bar{q}$
- return $(V, ar{q})$

Baseline Transformation

function $f_b(q, S)$
- $V \leftarrow q$  \hfill $\triangleright$ Initialize the view
- $V.select \leftarrow \emptyset$  \hfill $\triangleright$ Empty the select clause of $V$
- $V.select \leftarrow q.AttrList \cup \forall R \in q.\bowtie R.AttrList$
- $\bar{q} \leftarrow \text{SELECT COUNT(*) FROM } V$
- return $(V, \bar{q})$
end function

View Decorrelation

function $f_{dc}(V)$
- $DQ \leftarrow \emptyset$  \hfill $\triangleright$ Create Decorrelated query $DQ$
- $CQ \leftarrow \text{EXTRACT SQ}(q)$
- $DQ.select \leftarrow CQ.select + \bowtie R_c$
- $DQ.from \leftarrow CQ.from \setminus \{ "AS a" \}$
- $DQ.where \leftarrow \Phi$
- $DQ.groupby \leftarrow "\text{GROUP BY } R_c"$
- $DQ \leftarrow DQ + "AS a"$
- $V.select \leftarrow V.select + "A.c"$
- $V.from \leftarrow V.from + DQ$
- $V.where \leftarrow V.where + "\text{AND } c \ C"$  \hfill $\triangleright$ Update where condition of $q$
- $V.where \leftarrow V.where + "\text{AND } R = R_c"$
- return $V$
end function

Predicate Transfer

function $f_{pt}(V, \bar{q})$
- $\bar{q}.where \leftarrow V.where_{nj}$
- $V.where_{nj} \leftarrow \emptyset$
end function

VSelector uses QTransform (see Algorithm 1), a query transformation function that takes as input a query $q$ and returns a query-view pair $(\bar{q}, V)$. First, QTransform is applied on all queries of $Q$ and returns a set of query-view pairs $\{(\bar{q}, V)\}$. Then, all pairs with a common view are grouped together such that each view $V$ is associated with a set of transformed queries $Q_V$: the partial workload of view $V$. This is followed by a step of attribute pruning where each view $V$ retains only those attributes that appear in at least one query of $Q_V$. In Fig. 5.1 we see a
full execution of our VSELECTOR on a workload of 4 queries, resulting in views $V_1$ and $V_2$ with partial workloads $Q_{V_1} = \{\bar{q}_1, \bar{q}_2\}$ and $Q_{V_2} = \{\bar{q}_3, \bar{q}_4\}$ respectively.

QTransform is fully described in Algorithm 1, on a high level its functionality is summarized from 3 sequential steps: (a) the baseline query transformation $f_b$, (b) the view decorrelation step $f_{dc}$, and (c) the predicate transfer step $f_{pt}$, each described in Algorithm 1. More specifically, the baseline transformation $f_b$ creates a simple view $V$ that (a) contains all tuples that the input query accesses $q$ and (b) ensures that the view has all attributes accessed from the query. Next, the function $f_{dc}$ performs decorrelation [BMSU86] on a view $V$ by rewriting correlated subqueries of the view in terms of joins. Finally, $f_{pt}$ operates on input $(\bar{q}, V)$ and moves all filtering operations from the view $V$ to the query $\bar{q}$.

In the example of Fig. 5.1 we can see how QTransform operates on query $q_3$ which contains a correlated subquery in its WHERE clause.

which contains a correlated sub-query is transformed to the pair $(V_2, \bar{q}_3)$.

5.2 View Sensitivity Analysis

Computing the global sensitivity of a SQL view (lines 6-7 of Algorithm 1) is a hard problem [AFG16], as single changes in a base relation could affect a large (or even unbounded) number of records in the view. Moreover, complex privacy policies resulting in secondary private relations (see Definition 3.2.2), further complicate sensitivity estimation.

In this section we focus on simple privacy policies resulting only in a primary private relation in the schema and discuss complex policies in Section 5.3. Section 5.2.1 describes SENSCalc a rule-based algorithm that computes the constraint-oblivious down sensitivity of a view $V$ on a database instance $D$. Section 5.2.2 describes how to rewrite a view using truncation operators so that for simple privacy policies, the sensitivity output by SENSCalc is indeed the global sensitivity of the rewritten view.
Table 5.1: Update rules for sensitivity and max-frequency bounds. New rules are shaded.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Sensitivity Bound $\Delta_R(S)$</th>
<th>Max Frequency Bound $\hat{mf}(\mathcal{A}', S)$</th>
<th>Key Set $Keys(S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S = \pi_{\mathcal{A}}(R)$</td>
<td>$\Delta_R(R)$</td>
<td>$\hat{mf}(\mathcal{A}', R)$</td>
<td>${\mathcal{A}' \subseteq attr(S) \mid \mathcal{A}' \in Keys(R)}$</td>
</tr>
<tr>
<td>$S = \sigma_\phi(R)$</td>
<td>$\Delta_R(R)$</td>
<td>$\hat{mf}(\mathcal{A}', R)$</td>
<td>${\mathcal{A}' \subseteq attr(S) \mid \mathcal{A}' \in Keys(R)}$</td>
</tr>
<tr>
<td>$S = \gamma_{\mathcal{A}}(R)$</td>
<td>$\Delta_R(R)$</td>
<td>$\hat{mf}(\mathcal{A}', R)$</td>
<td>${\mathcal{A}' \subseteq attr(S) \mid \mathcal{A}' \in Keys(R)}$</td>
</tr>
<tr>
<td>$S = \gamma_{\mathcal{A}}^{COUNT}(R)$</td>
<td>$\Delta_R(R)$</td>
<td>$\hat{mf}(\mathcal{A}', R)$</td>
<td>${\mathcal{A}' \subseteq attr(S) \mid \mathcal{A}' \in Keys(R)}$</td>
</tr>
<tr>
<td>$S = R_1 \bowtie_{\mathcal{A}_1 = \mathcal{A}<em>2} R_2$ or $S = R_1 \bowtie</em>{\mathcal{A}_1 = \mathcal{A}_2} R_2$ where $\mathcal{A}_1, \mathcal{A}_2$ are from $S$</td>
<td>General case $\Delta_R(R)$</td>
<td>$\max(\hat{mf}(\mathcal{A}_1, R_1) \cdot \Delta_R(R_2), \hat{mf}(\mathcal{A}_2, R_2) \cdot \Delta_R(R_1))$</td>
<td>${\mathcal{A}' \in Keys(R_1) } \cup {\mathcal{A}' \in Keys(R_2) }$</td>
</tr>
<tr>
<td>$S = \tau_{\mathcal{A}, \xi}(R)$</td>
<td>$k \cdot \hat{\Delta}_R(R)$</td>
<td>$\min{ k, \hat{mf}(\mathcal{A}', R) } \text{ if } \mathcal{A} \subseteq \mathcal{A}'$</td>
<td>${\mathcal{A}' \subseteq S \mid \mathcal{A}' \in Keys(R)}$</td>
</tr>
</tbody>
</table>

$V'$ (see Theorem 5.2.1). Section 5.2.3 presents a DP method for learning thresholds needed for truncation operators.

We assume w.l.o.g. that a view $V$ is expressed in relational algebra. This expression can be viewed as a tree, where internal nodes are algebra operators and the leaf nodes are base relations of $S$. First, we propose SensCalc a rule-based algorithm for computing a bound on the sensitivity of a view (Section 5.2.1). We also bound the sensitivity of join operations by a query rewrite – by adding targeted truncation operators on the query plan of a view (Section 5.2.2). Our main theoretical result of this section, Theorem 5.2.1, shows that any view $V$ can be rewritten to a view $V'$ such that the sensitivity calculator returns a bound on the global sensitivity of $V'$. 

39
In the sequel (Section 5.3) we use our results to extend PrivSQL so that it can automatically handle complex privacy policies.

5.2.1 Sensitivity Calculator

Sensitivity estimation in PrivSQL is performed by the SensCalc module. This module computes the constraint-oblivious down sensitivity, a sensitivity variant that captures the maximum change caused by removing any one tuple from the primary private relation \( R \).

**Definition 5.2.1** (Constraint-Oblivious Down Sensitivity). Given schema \( S \) and a privacy policy \((R, \epsilon)\), the constraint-oblivious down sensitivity of \( V \) given \( D \) w.r.t. \( R \), denoted by \( \Delta_R(V, D) \), is defined as the maximum number of rows that change in \( V \) when removing a row from \( R \).

\[
\Delta_R(V, D) = \max_{r \in \text{dom}(R)} V(D) \Delta V(D - \{r\}),
\]

where \( D - \{r\} \) means removing tuple \( r \) from instance \( D \).

In the case of simple privacy policies, the constraint-oblivious down sensitivity is equivalent to the down sensitivity (defined in Section 3.2 Eq. (5.2)), i.e., for any simple policy \( P \) and any \( V \):

\[
\Delta_R(V, D) = \Delta_C(R, V, D).
\]

Combined with truncation rewrites described later, the sensitivity output by SensCalc will be the right global sensitivity for simple policies.

SensCalc is a recursive rule-based sensitivity calculator that takes as input \( V \), schema \( S \), and a relation \( R \) designated as the primary private relation. It also has access to \( \hat{mf} \), a function that provides bounds on the maximum frequency \( mf \) of any attribute combination of the base relations in \( V \). The final result is \( \hat{\Delta}_R(V, \hat{mf}) \), as it depends on the bounds supplied from \( \hat{mf} \) – when clear from context we write \( \hat{\Delta}_R(V) \).

Given an input view \( V \) and \( \hat{mf} \), the sensitivity calculator computes \( \hat{\Delta}_R(V, \hat{mf}) \) by a recursive application of the rules in Table 5.1 to each subexpression \( S \) of \( V \).
The bounds at the base relations are as follows: the sensitivity bounds $\Delta_R(R) = 1$ and $\Delta_R(R) = 0$ for $R \in S \setminus \{ R \}$ and the max-frequency bounds are supplied by $\hat{mf}$. In Table 5.1 we summarize the rules of SensCalc. Operators such as `PROJECT`, `SELECT`, and `GROUPBY` do not increase the sensitivity bound of their input relation, while `GROUPBY-COUNT` doubles it. `EQUIJOIN` results in relations with higher sensitivity bounds compared to its inputs. In terms of the $mf$ bounds, most unary operators shown in Table 5.1 have unchanged $mf$. Note that we restrict the `EQUIJOIN` operator to join on attributes from the base relations in $S$. The last row refers to a `truncation operator`, which is described in Section 5.2.2.

These rules are similar to those of elastic sensitivity [JNS18], but with some key differences that allow for a tighter sensitivity analysis. SensCalc uses additional rules using `keys`, as shown in the last column of Table 5.1. The new rules keep track of key constraints through operators. This allows the addition of new rules for joins on key attributes that permit lower sensitivity bounds than a standard join, as illustrated in the following example.

**Example 2 (Sensitivity Calculation).** Consider calculating the sensitivity of $V_2$ from Fig. 5.1 under Person policy. A relational algebra expression for view $V_2$ is (Fig. 5.2
V₂ has a row for each person reporting the person’s race, relp, and size of their household. SENS CALC initializes $\Delta_R(\text{Person})$ to 1 and applies the rules of Table 5.1 bottom up. First the GROUPBY-COUNT operator is processed, resulting in $S = \gamma_{hid}(\text{Person})$ with $\Delta_R(S) = 2 \cdot \Delta_R(\text{Person}) = 2$ and $S$ has hid as a key. Next, the EQUJOIN operator is processed, joining on key hid of $S$, producing $S_{\bowtie hid} = \text{Person} \bowtie hid S$ with: $\Delta_R(S_{\bowtie hid}) = F \cdot \Delta_R(S) + \Delta_R(\text{Person}) = F \cdot 2 + 1$ where $F = \hat{mf}(hid, \text{Person})$. Note that without the “Join on key” rule, the bound would be $(F \cdot 3 + 2)$. This difference is only exacerbated for views with more joins. Last, the PROJECTION operator is processed, leaving the bound unchanged.

Given $D$, $V$ and upper bounds on max-frequency $\hat{mf}$, we can show that $\Delta_R(V, \hat{mf})$ calculated by SENS CALC is an upper bound on $\Delta_R(V, D)$, and thus an upper bound on the down sensitivity $\Delta_C(V, D)$ for simple policies.

5.2.2 Bounding Sensitivity via Truncations

As shown in Example 2, the sensitivity bounds produced by the SENS CALC can be dependent on the max-frequency bounds on base relations. We now show how to add truncation operators to the view expression. These operators delete tuples that contain an attribute combination appearing in a join and whose frequency exceeds a truncation threshold $k$ specified in the operator. The sensitivity will no longer depend on max-frequencies but rather on the thresholds. If thresholds are set in a data-independent manner or using a DP algorithm, then we show that the sensitivity computed by SENS CALC is indeed a bound of the global sensitivity.

**Definition 5.2.2** (Truncation Operator). The truncation operator $\tau_{A,k}(R)$ takes in a relation $R$, a set of attributes $A \subseteq \text{attr}(R)$ and a threshold $k$ and for all $a \in \text{dom}(A)$,
Algorithm 2 Truncation Rewrite \((V, R, k)\)

1: Initialize \(V^\tau \leftarrow V\)

2: for every path \(p_l\) from leaf relation \(R_l\) to root in \(V\) do

3: for every \(R_1 \bowtie_{A_1=A_2} R_2\) on \(p_l\), where \(A_1 \subseteq \text{attr}(R_l)\) do

4: \> (semijoin is also treated as a special equijoin)

5: if \(A_1 \notin \text{Keys}(R_1)\) and \(R\) is a base relation of \(R_2\) then

6: \> \(k \leftarrow k_{A_1}\)

7: \> Insert \(\tau_{A_1,k}(R_l)\) above \(R_l\) in \(V^\tau\)

8: \> \(A \leftarrow A \cup (A_1)\)

9: end if

10: end for

11: end for

12: Return \(V^\tau\)

if \(f(a, A, R) > k\), then any \(r\) from \(R\) with \(r[A] = a\) is removed.

Truncation rewrite (see Algorithm 2) adds truncation operators to \(V\) and forms a new query plan \(V^\tau\). The algorithm takes as input a view \(V\), a primary private relation \(R\), and a vector of truncation thresholds \(k\), indexed by the attribute subset to which the threshold applies. It traverses every path \(p_l\) from relation \(R_l\) to the root operator and every join \(R_1 \bowtie_{A_1=A_2} R_2\) on this path. If one of the join attributes is from \(R_l\)—say \(A_1 \subseteq R_l\)—and \(A_1\) is not a key for \(R_1\) and the primary private table \(R\) appears as a base relation in the expression \(R_2\), then we insert \(\tau_{A_1,k}(R_l)\) above \(R_l\) in \(V^\tau\). The rules of \textsc{SensCalc} for the truncation operator can be found on Table 5.1.

In terms of the maximum frequency bound, it is at most \(k\) for any \(A' \supseteq A\).

Example 3. Fig. 5.2 (right) shows the truncation operators are inserted before \texttt{Person} relation. The truncation operators cut down the maximum frequency of hid to \(k\) so that the sensitivity bound can be bounded by \(3k\), even when \(\hat{mf}\) for household id in \texttt{Person} is unbounded. In this case, \(\hat{\Delta}_R(S_\infty) = k \cdot \hat{\Delta}_R(\gamma_{\text{COUNT}}(\texttt{Person})) + \hat{\Delta}_R(\tau_{\text{hid,k}}(\texttt{Person})) = k \cdot 2 + k = 3k\).

After truncation rewrite is applied, the estimated sensitivity no longer depends on \(\hat{mf}\), but rather on the truncation thresholds. If the thresholds are set in a data independent manner, or using a DP algorithm (as discussed in Section 5.2.3) we can
show that the sensitivity output by SensCalc on $V^\tau$ is the global sensitivity for simple policies.

**Theorem 5.2.1.** Consider a schema $\mathcal{S} = (R_1, \ldots, R_k)$ with foreign constraints $\mathcal{C}$, and simple privacy policy $(R, \varepsilon)$. For any $V$, let $V^\tau$ denote the truncation rewrite of $V$ using a fixed set of truncation thresholds $k$ (Algorithm 2). The global sensitivity of $V^\tau$ is bounded by SensCalc:

$$\Delta^C_R(V^\tau) = \Delta_R(V^\tau) \leq \hat{\Delta}_R(V^\tau).$$

Let $M$ be $\varepsilon_v$-differentially private algorithm that runs on $V^\tau(D)$. Then $M$ satisfies $P_{\varepsilon_v}$-DP with $P_V = (R, \varepsilon_v \cdot \hat{\Delta}_R(V^\tau))$.

**Proof.** Part I: Let $\widehat{mf}^\infty$ denote unbounded max frequencies: $\widehat{mf}^\infty(A, R) = \infty$ for all $A \subseteq \text{attr}(R)$ and for all $R \in \mathcal{S}$.

For any $D \in \text{dom}(\mathcal{S}, \mathcal{C})$,

$$\Delta^C_R(V^\tau, D) = \Delta_R(V^\tau, D) \quad \text{For simple policies}$$

$$\leq \hat{\Delta}_R(V^\tau, mf) \quad \text{(by Lemma 5.2.1)}$$

$$= \hat{\Delta}_R(V^\tau, \widehat{mf}^\infty) \quad \text{(by Lemma 5.2.2)}$$

$$= \hat{\Delta}_R(V^\tau) \quad \text{(simplified notation)}$$

Because the above bound holds for all $D$ it also bounds the global sensitivity.

Part II: If we run an $\varepsilon_v$-differentially private mechanism $M$ on $V^\tau(D)$, then for any pair $(S, S')$ that differ in $k$ records, where $S, S'$ are possible output of $V^\tau(\cdot)$, we have $|\ln(\frac{M(S)}{M(S')})| \leq (\varepsilon_v \cdot k)$. For any $D, D'$ neighbors with $R$ is the primary private relation, $V^\tau(D)$ and $V^\tau(D')$ differ by at most by $\hat{\Delta}_R(V^\tau)$. Therefore, $M$ satisfies $P_{\varepsilon_v}$-DP with $P_V = (R, \varepsilon_v \cdot \hat{\Delta}_R(V^\tau))$. 

$\square$
The truncation rewrite introduces bias: i.e., $\exists D, V(D) \neq V^r(D)$. However, the global sensitivity computed after truncation is usually much smaller reducing error due to noise. We empirically measure the effect of truncation bias in Section 7.1.4. Our truncation methods are related to Lipschitz extension techniques which also tradeoff bias for noise typically by truncating the data. Existing methods apply to specific queries on graphs [HLMJ09, KRSY11, KNRS13, DLL16, DZBJ18] or only on monotone queries [CZ13]. Our technique applies to general relational data and more complex queries.

To proof of Theorem 5.2.1 is supported by the following two lemmas that show given a view $V$, SensCalc calculates a upper bound on the constraint-oblivious down sensitivity of $V$ on input $D$.

**Lemma 5.2.1.** Consider an acyclic schema $S = (R_1, \ldots, R_k)$ with foreign constraints $C$, a single private relation $R \in S$, and no secondary private relations. For all views $V$, inputs $D$, base tables $S$, and all $A \subseteq \text{attr}(S)$, if $mf(A, S) \leq \hat{mf}(A, S)$ then: $\Delta_R(V, D) \leq \hat{\Delta}_R(V, \hat{mf})$.

**Proof.** The rules presented in Table 5.1 with white background are first proposed in [JNS18]. The new rule on joining on a key attribute is as follows. Let $S = R_1 \bowtie_{A_1=A_2} R_2$ an equijoin where $A_1$ is a key attribute on $R_1$. The removal of a single tuple can affect $mf(A_2, R_2)\hat{mf}(R_1)$ tuples in $S$ from the influence of $R_1$. However, $A_1$ is a key on $R_1$ with max frequency 1, that means that the influence of $R_2$ is $\hat{mf}(R_2)$. Hence the overall sensitivity of $S$ is bounded by $mf(A_2, R_2)\hat{mf}(R_1) + \hat{mf}(R_2)$.

The new rule on the proposed truncation operator is as follows. Let $S = \tau_{A,k}(R)$ a truncation on relation $R$ for attribute $A$, at value $k$. This means that $S$ will contain tuples with value for $A$ at most $k$. Let $R'$ a neighboring instance: $R' = R - \{t\}$, s.t. $v = t.A$ has multiplicity $k + 1$, and $S' = \tau_{A,k}(R')$. It is then obvious that $S'$ has $k$ less tuples than $S$ since truncation in $R$ does not affect $k$ tuples with value $v$. Hence
Algorithm 3 LearnThreshold $(D, V^\tau, \theta, \epsilon_{mf})$

1: Traverse operators in $V^\tau$ from leaf to root and add each truncation operator to $T$ if it is not in the list.
2: for $\tau_{A,k}(R) \in T$ do
3: \[ q'_i \leftarrow \text{sub-tree at } \tau_{A,k}(R) \in V^\tau \] \[ \triangleright \text{Truncate at } k = i \]
4: \[ Q \leftarrow \{(\frac{|q'_i|-|R_d|}{i} | i = 1, 2, \ldots) \} \]
5: Set $i \leftarrow \text{SVT}(D, Q, 0, \epsilon_{mf}/|T|)$ as the truncation threshold for $\tau_{A,k}(R)$
6: end for

the sensitivity of $\tau_{A,k}(R)$ is $\hat{mf}(R)k$.

We show in Lemma 5.2.2 that truncation eliminates the need for tight bounds on max frequencies.

**Lemma 5.2.2.** For any $V$, let $V^\tau$ denote the truncation rewrite of $V$ using a fixed set of truncation thresholds $k$. Let $\hat{mf}^\infty$ denote unbounded max frequencies: $\hat{mf}^\infty(A, R) = \infty$ for all $A \subseteq \text{attr}(R)$ and for all $R \in S$. For any $\hat{mf}$ such that $mf(A, S) \leq \hat{mf}(A, S)$ for all base relations $S$ of $V$ and all $A \subseteq \text{attr}(S)$: $\hat{\Delta}_R(V^\tau, \hat{mf}) = \hat{\Delta}_R(V^\tau, \hat{mf}^\infty)$

**Proof.** Algorithm 2 adds truncation operators on top of base relations that participate in joins (later in the tree of $V$). Since SENSICALC works in a bottom-up fashion, this removes the dependency of SENSICALC on the true max frequencies of the base tables. Thus, $\hat{\Delta}_R(V^\tau, \hat{mf}) = \hat{\Delta}_R(V^\tau, \hat{mf}^\infty)$.

Hence, the global sensitivity of the rewritten query $\Delta^G_R(V^\tau)$ is upper bounded by $\hat{\Delta}_R(V^\tau)$ outputted by SENSICALC.

5.2.3 Learning Truncation Thresholds

In Section 5.2.2 we described how we use truncation operators to bound the computed view sensitivity. From Definition 5.2.2 we observe that the threshold $k$ plays a crucial role in the function of the truncation operators.

Setting this threshold can be done independently of the underlying data (e.g., based on public knowledge), or in a privacy-preserving, data dependent fashion. We
opt for the latter and propose **LearnThreshold** (see Algorithm 3), an algorithm that given a specific data input, outputs a vector of thresholds indexed by the truncation operator they correspond to.

In Algorithm 3 we fully describe **LearnThreshold**. It takes as input privacy parameter \( \epsilon_{mf} \) and \( \theta \), the fraction of rows we would like to preserve in the truncated relation. **LearnThreshold** works in a bottom-up manner to identify the ordered list \( T \) of unique truncation operators in \( V^\tau \). For each truncation operator \( \tau_{A,k}(R) \), let \( q'_i \) be the sub-query rooted at the operator if truncation threshold \( k \) is set to be \( i \). We consider a stream of queries \( Q = \{ q_i \mid i = 1, 2, \ldots \} \), where \( q_i = (|q'_i(D)| - |R| \cdot \theta)/i \) measures whether \( \theta \) fraction of \( R \) can be preserved if truncating \( R \) at threshold \( i \). The sensitivity of \( q_i \) is bounded by the sensitivity of \( R \), which in turn is bounded since the **LearnThreshold** operates bottom-up. We apply the *sparse vector technique* [DR14] which returns the first \( i \) such that \( q_i(D) > 0 \) with the given privacy budget \( \epsilon_{mf}/|T| \). Each call of SVT incurs privacy loss \( \epsilon/|T| \), thus by sequential composition the overall privacy loss incurred by **MaxFreqCalc** is bounded by \( \epsilon_{mf} \).

### 5.3 Handling Complex Policies

We now shift our focus on computing view sensitivity for *complex privacy policies*. Recall that under complex privacy policies, neighboring databases differ in the primary private relation as well as other secondary private relations (see Fig. 3.1c for reference). Due to this, the constraint oblivious down sensitivity is not the same as the down sensitivity (i.e., \( \Delta_k(V,D) \neq \Delta_k^c(V,D) \)). Moreover, removing a row in the primary private relation might result in an unbounded number of rows deleted in secondary private relations – e.g., under **Household** policy the maximum change in **Person** is unbounded in the absence of external information. Truncation operators discussed previously only limit the frequencies of attributes involved in joins, but not
the change in secondary private relations.

We first present the semijoin rewrite that transforms view $V$ into $V^\ominus$ so that the sensitivity computed by \textsc{SensCalc} on $V^\ominus$ equals its down sensitivity (i.e., $\Delta_S(V^\ominus, D) = \Delta_S(V^\ominus, D)$). For example, consider the view $V_1$ from Fig. 5.1 under Household policy where Person is a secondary private relation. In that example, removing a tuple from Household will result in removing multiple tuples from Person, thus affecting the sensitivity of $V_1$.

To address these challenges, we introduce the notion of transitive referral and deletions, which allows reasoning about neighboring databases. We also propose an additional view rewriting operation, such that even for complex privacy policies executing the sensitivity calculation algorithm of Section 5.2.1 on the rewritten view automatically computes the correct sensitivity bounds of the original view.

**Transitive Referral and Deletion:** If $S.A_{fk} \rightarrow R.A_{pk}$ is a foreign key constraint, deleting a row $r$ in relation $R$ results in the cascading deletion of all rows $s \in S$ such that $s[A_{fk}] = r[A_{pk}]$. Furthermore, if $T.A'_{fk} \rightarrow S.A'_{pk}$, then the deletion of record $s \in S$ can recursively result in the deletion of records in $T$. We define this property as transitive referral.

**Definition 5.3.1** (Transitive Referral). A relation $S$ transitively refers to a relation $R$ through foreign keys if there exists a relation $T$ such that $S.A \rightarrow T.B$ and $T$ transitively refers to relation $R$ through foreign keys. Moreover, a row $s \in S$ transitively refers to a row $r \in R$ if there is a row $t \in T$ such that $s \rightarrow t$ and $t$ transitively refers to $r$. If $s$ transitively refers to $r$, we denote that $s \rightarrow r$.

A schema is acyclic if no relation in it transitively refers to itself. We now propose a method of deriving neighboring databases under acyclic schemas.

**Theorem 5.3.1** (Transitive Deletion). Given an acyclic schema $S = (R_1,\ldots,R_k)$ with foreign key constraints $C$, and a privacy policy $(R_i, \epsilon)$. For $D \in \text{dom}(S, C)$ and
\( r \in D_i \), we denote \( \ominus_C(D_i, (r, R_i)) = (D_1^\ominus, D_2^\ominus, \ldots, D_k^\ominus) \), where \( D_j^\ominus = D_j - \{ t | t \in D_j, t \rightarrow r \} \). Then we have:

\[
\ominus_C(D_i, R_i) = \bigcup_{r \in D_i} \ominus_C(D_i, (r, R_i)).
\]

**Proof.** First, we show that for all \( r \in D_i \), \( \ominus_C(D_i, (r, R_i)) \in \ominus_C(D_i, R_i) \). As \( r \in D_i \) and \( D_i^\ominus = D_i - \{ r \} \), we have \( r \notin D_i^\ominus \). For any \( R_j \) and for all \( R_p \) that is referred by \( R_j \): \( D_j^\ominus \times D_p^\ominus = D_j^\ominus \). Let the following definitions:

- \( \rightarrow_X(D_j, r) = \{ t \in D_j | t \rightarrow r \} \),
- \( \rightarrow_X(D_j, D_p, r) = \{ t \in D_j | \exists s \in D_p, t \rightarrow s \land s \rightarrow r \} \).

Then, we have:

\[
D_j^\ominus \times D_p^\ominus = (D_j - \rightarrow_X(D_j, r)) \times (D_p - \rightarrow_X(D_p, r))
\]

\[
= D_j - \rightarrow_X(D_j, r) - \rightarrow_X(D_j, D_p, r) + \rightarrow_X(D_j, D_p, r) = D_j^\ominus
\]

Hence, \( D^\ominus \) satisfies all the foreign key constraints \( Q \) by Definition 2.2.1.

Last, suppose there exists \( D'' \) that satisfies \( Q \) and \( D \sqsubseteq D'' \sqsubseteq D^\ominus \). Then \( \exists j \), \( D_j \supseteq D_j' \supseteq D_j^\ominus = (D_j - \{ t \in D_j | t \rightarrow r \}) \). Thus, there exists \( s \in D_j' \) s.t. \( s \rightarrow r \), which leads to a contradiction: \( r \notin D_i^\ominus \).

Secondly, we show that if \( D' \in \ominus_C(D_i, R_i) \), then there exists \( r \in D_i \) such that \( D' = (D_1^\ominus, D_2^\ominus, \ldots, D_k^\ominus) \), where \( D_j^\ominus = D_j - \{ t | t \in D_j, t \rightarrow r \} \). Suppose this is not true, i.e., exist a \( D_j' \neq D_j^\ominus \): (i) exist \( t \in D_j' \) such that \( t \rightarrow r \), or (ii) exist \( t \in (D_j - D_j') \) such that \( t \not\rightarrow r \). The first case will imply \( D' \) conflicts \( C \) as \( r \notin D_i \). The second case will either conflict the minimality condition (exist \( D'' \) that satisfies \( C \) and \( D \sqsubseteq D'' \sqsubseteq D' \)) or implies the schema contains cycle, which is again a contradiction, thus concluding the proof.

Based on this theorem, the down sensitivity of a view (defined in Definition 3.2.4) can be expressed as:

\[
\Delta_C^r(V, D) = \max_{r \in \text{dom}(R)} V(D) \Delta V(\ominus_C(D_i, (r, R_i))). \tag{5.2}
\]
**Semijoin Rewrite:** Our proposed rewrite works in two steps. First, it replaces every secondary private base relation $R_j$ in $V$ with a semijoin expression (Eq. (5.3)) that makes explicit the transitive dependence between the primary private relation $R$ and $R_j$. The resulting expression $V^x$ is such that $V(D) = V^x(D)$. Moreover, the down sensitivity is now correct $\Delta_R(V^x, D) = \Delta^c_R(V^x, D)$ since transitive deletion is captured by the semijoin expressions.

Second, to handle the high sensitivity of secondary private base relations, we add truncation operations using (Algorithm 2) to the semijoin expressions and transform $V^x$ to $V^\ominus$. More formally, Recall that the sensitivity calculator is based on the constraint-oblivious down sensitivity from Definition 5.2.1, which is different from the down sensitivity in Definition 3.2.4 when there are multiple private relations. To fill the gap, we propose semijoin rewrite that captures the transitive deletion of a single row in the primary private relation, so the sensitivity calculator can still output the correct sensitivity given multiple private relations.

**Definition 5.3.2 (Semijoin Rewrite).** The semijoin rewrite:

1) takes as input $V$ and transforms it into $V^x$ such that $V^x$ is identical to $V$ except that each base relation $R_j$ of $V$ is replaced with $R_j^x$, which is recursively defined as:

$$R_j^x = \begin{cases} 
R_j, & \text{if } R_j = R \\
((R_j \times R_{p(j)}^{x_1}) \times R_{p(j)}^{x_2}) \ldots \times R_{p(j)}^{x_\ell}) & \text{else}
\end{cases} \tag{5.3}$$

where each relation $S \in \{R_{p(j)}^{x_1}, R_{p(j)}^{x_2}, \ldots, R_{p(j)}^{x_\ell}\}$ is such that: (a) $R_j$ refers to $S$, and (b) $S = R$ or transitively refers to the primary private relation $R$ through foreign keys.

2) It transforms $V^x$ into $V^\ominus$ such that $V^\ominus$ is identical to $V^x$ except that each $R_j^x$ is replaced by $R_j^\ominus$ by running Algorithm 2, which is the truncation rewrite of $R_j^x$.

This rewrite eliminates the need to consider foreign key constraints and bounds
the sensitivity of each replaced expression.

**Lemma 5.3.1.** Given an acyclic schema $S$ with foreign key constraints $C$, privacy policy $P = (R, \epsilon)$, and a view $V$. Let $V^\times$, $V^\ominus$ be as defined in Definition 5.3.2. Then, for any database instance $D \in \text{dom}(S, C)$, we have $V(D) = V^\times(D)$ and the down sensitivity of $V^\ominus$ equals the constraint-oblivious down sensitivity of $V^\ominus$:

\[
\Delta_R^C(V^\ominus, D) = \Delta_R(V^\ominus, D)
\] (5.4)

**Proof.** First, it is easy to see that $V(D) = V^\times(D)$ for $D \in \text{dom}(S, Q)$, by the definition of a foreign key (Definition 2.2.1) as $R^\ominus_j(D) = R_j(D)$ for all $R_j$ in the schema. We denote $R(D)$ as the instance of $R$ given the database $D$ and $R$ is the relation schema.

Next, we need to show that for any $r \in \text{dom}(R)$, for any given $D \in \text{dom}(S, Q)$, $V(D^\ominus) = V^\times(D')$, where $D' = D - \{r\}$ and $D^\ominus = \ominus_C(D, (r, R))$, by proving that for any $R_j$ in the schema,

\[
R^\ominus_j(D - \{r\}) = D^\ominus_j.
\] (5.5)

where $D^\ominus_j = D_j - \{t | t \in D_j, t \rightarrow r\}$

Let $R_i$ be the primary private relation $R$. Let $\bar{X}(D_j, r) = \{t \in D_j | t \rightarrow r\}$, $\bar{X}(D_j, D_p, r) = \{t \in D_j | \exists s \in D_p, t \rightarrow s \land s \rightarrow r\}$.

**Base step:** When $j = i$, Eq. (5.6) is true as $R^\ominus_i = R_i$.

**Induction:** Suppose that given $R_j$, all $R_{p(j)} \in \{R_{p(j)}, \ldots, R_{p(j)}\}$ such that (a) $R_j$ refers to $R_{p(j)}$, and (b) $R_{p(j)}$ transitively refers to $R_i$ through foreign keys, satisfy Eq. (5.6), i.e.,

\[
R^\ominus_{p(j)}(D - \{r\}) = D_{p(j)} - \bar{X}(D_{p(j)}, r).
\] (5.6)

We want to show $R^\ominus_j$ satisfies Eq. (5.6). We abuse the usage of $R^\ominus_j$ as $R^\ominus_j(D - \{r\})$,
\( R_j \) as \( R_j(D - \{ r \}) \).

\[
(((R_j \bowtie R_{p(j)_1}^\ominus) \bowtie R_{p(j)_2}^\ominus) \ldots \bowtie R_{p(j)_\ell}^\ominus)
\]
\[
= (((R_j \bowtie (R_{p(j)_1} - \overrightarrow{X}(R_{p(j)_1}, r))) \bowtie R_{p(j)_2}^\ominus) \ldots \bowtie R_{p(j)_\ell}^\ominus)
\]
\[
= (((R_j - \overrightarrow{X}(R_j, R_{p(j)_1}, r)) \bowtie (R_{p(j)_2} - \overrightarrow{X}(R_{p(j)_2}, r))) \ldots \bowtie R_{p(j)_\ell}^\ominus)
\]
\[
= ((R_j - \overrightarrow{X}(R_j, R_{p(j)_1}, r)) \cup \overrightarrow{X}(R_j, R_{p(j)_2}, r)) \ldots \cup \overrightarrow{X}(R_j, R_{p(j)_\ell}, r)
\]
\[
= R_j - \overrightarrow{X}(R_j, r)
\]  

(5.7)

This gives us \( V(D^\ominus) = V^\prec(D') \). Therefore, we can have \( V(D) \Delta V(D^\ominus) = V^\prec(D) \Delta V^\prec(D') \).

Lemma 5.3.1 justifies the use of the simpler notion of sensitivity employed by SensCalc in Section 5.2.1. Note that, for some \( D \), \( V(D) \neq V^\ominus(D) \) due to the truncation rewrite.

**Putting it all together:** Given a view \( V \), we first apply Algorithm 2 to \( V \) to add truncation operators to the primary private relation \( R \) and obtain \( V^\tau \). Then we run semijoin rewrite in Definition 5.3.2 to get \( V^{\tau,\ominus} \).

As the second step of semijoin rewrite introduces extra truncation operators into the query plan, existing truncation operators may become redundant, in which case we keep ones closest to the base relation. The following example shows the entire procedure of a view rewrite.

**Example 4.** Recall the query plan \( V \) and its truncation rewrite \( V^\tau \) from Fig. 5.2. Under the **Household** policy, **Person** is a secondary private relation. As shown in Fig. 5.3 the semijoin rewrite will replace the **Person** relations in \( V^\tau \) with a semijoin...
between Person and Household. Truncation operators are also added to bound the sensitivity of the Person table to get $V^\tau\ominus$. Note that the truncation operator in $V^\tau$ is redundant in $V^\tau\ominus$ and removed since the semijoin rewrite introduces the same truncation operator on Person. After truncation rewrite with threshold $k$, SensCalc outputs a bound on the global sensitivity: $\hat{\Delta}_R(V'_2) = 2 \cdot k^2 + k$.

Theorem 5.3.2 shows that after applying the truncation and semijoin rewrites the sensitivity of $V^\tau\ominus$ output by SensCalc is the global sensitivity. Proof follows from Theorem 5.2.1 and Lemma 5.3.1.

**Theorem 5.3.2.** Given an acyclic schema $S = (R_1, \ldots, R_k)$ with foreign constraints $C$, and $R \in S$. For any $V$, let $V^\tau\ominus$ denote $V$ after applying both the truncation rewrite (Algorithm 2) and the semijoin rewrite (Definition 5.3.2), where the truncation thresholds are $k$ and are fixed. The global sensitivity of $V^\tau\ominus$ is bounded:

$$\Delta_R^C(V^\tau\ominus) \leq \hat{\Delta}_R(V^\tau\ominus).$$

Let $M$ be $\epsilon_v$-differentially private algorithm that runs on $V^\tau\ominus(D)$. Then $M$ satisfies $P_V$-DP with $P_V = (R, \epsilon_v \cdot \hat{\Delta}_R(V^\tau\ominus))$.  

53
The proof follows from applying the argument of Theorem 5.2.1 with the addition of Lemma 5.3.1. More specifically:

Proof. Part I: Let $\hat{mf}^\infty$ be as defined in Lemma 5.2.2. For any $D \in \text{dom} (S, C)$,
\[
\Delta^C_R (V^{\tau, \ominus}, D) \\
= \Delta^C_R (V^{\tau, \ominus}, D) \quad \text{(by Lemma 5.3.1)} \\
\leq \hat{\Delta}^C_R (V^{\tau, \ominus}, mf) \quad \text{(by Lemma 5.2.1)} \\
= \hat{\Delta}^C_R (V^{\tau, \ominus}, \hat{mf}^\infty) \quad \text{(by Lemma 5.2.2)} \\
= \hat{\Delta}^C_R (V^{\tau, \ominus}) \quad \text{(simplified notation)}
\]
Because the above bound holds for all $D$, it also bounds the global sensitivity.

Part II: If we run an $\epsilon_V$-differentially private mechanism $M$ on $V^{\tau, \ominus}(D)$, then for any pair $(S, S')$ that differ in $k$ records, where $S, S'$ are possible output of $V^{\tau, \ominus}(\cdot)$, we have $|\ln \left( \frac{M(S)}{M(S')} \right) | \leq (\epsilon_V \cdot k)$. For any $D, D'$ neighbors with $R$ is the primary private relation, $V^{\tau, \ominus}(D)$ and $V^{\tau, \ominus}(D')$ differ by at most by $\hat{\Delta}^C_R (V^{\tau, \ominus})$. Therefore, $M$ satisfies $P_V$-DP with $P_V = (R, \epsilon_V \cdot \hat{\Delta}^C_R (V^{\tau, \ominus}))$. 

5.4 Private Synopsis Generator

The PrivSynGen module produces a private synopsis of a single materialized view on the sensitive data. The input to PrivSynGen is a materialized view $V(D)$, a set of linear (on $V$) queries $Q_V$, and a privacy budget $\epsilon_V$. Its output is $\tilde{D}_V$, an $\epsilon_V$-DP synopsis of $V(D)$, w.r.t. the materialized view $V(D)$.

One consideration is whether to release synthetic tuples or vectors of counts. The former is efficient in terms of representation – the vector form encodes one count for every possible tuple in the cross product of the domains of the attributes in the table, and is thus exponential in the number of attributes. However, the latter allows maintaining fractional counts, which leads to lower error. In addition, vector form
allows the use of linear algebra based inference methods to reason across multiple independent noisy releases, which can help answer queries not present in \( Q \). As noted earlier there is no constraint on the type of a synopsis returned from PrivSynGen. For example a synopsis could be a set of tuples drawn from a distribution, or a statistical summary of \( V(D) \).

This component is probably the most well understood as it is an instance of a common problem studied in the DP literature – answering a set of linear queries on a single table [ZCP+14, HMM+16, MMHM18]. Furthermore, synopsis generators can be \textit{workload aware} or \textit{workload agnostic} depending on whether they optimize their output w.r.t. a set of linear queries \( Q_V \).

We use both workload-agnostic and workload-aware instances of PrivSynGen, returning a vector of counts. More specifically, we use: \textit{W-NNLS}, a workload-aware version of non-negative least squares inference [LMH+15], and the workload-agnostic algorithms \textsc{Identity} and \textsc{Part}, the latter of which performs the partitioning step of the DAWA algorithm [LHMW14].

Let \( \mathbf{x} \) the vector form describing a materialized view \( V(D) \), each cell of \( \mathbf{x} \) encodes a different element of the cross-domain of the attributes in \( V \) and \( x_i \) is the count of tuples in \( V(D) \) with value equal to that decoding.

**Identity** The first synopsis generator we consider is \textsc{Identity} a \textit{workload agnostic} method, which takes as input the vector form of the materialized view \( \mathbf{x} \) and outputs \( \tilde{\mathbf{x}} = \mathbf{x} + \xi \), where each \( \xi_i \) is drawn i.i.d. from \textsc{Laplace}(0, \epsilon_V), a Laplace distribution with mean 0 and scale \( \epsilon_V \).

**Workload** We now describe \textit{W-NNLS} (Workload non-negative least squares)[ZMK+18], a \textit{workload aware} technique that first computes \( \mathbf{y} \) the true answers of a workload on \( \mathbf{x} \), then adds noise to them and lastly uses non-negative least squares to produce a private estimate of \( \mathbf{x} \). More specifically, let \( \mathbf{W} \) be the vector form of a query workload.
Table 5.2: Instantiations of BUDGETALLOC.

<table>
<thead>
<tr>
<th>Name</th>
<th>( \lambda ) parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAIVE</td>
<td>( \lambda_V = 1/</td>
</tr>
<tr>
<td>WSize</td>
<td>( \lambda_V =</td>
</tr>
<tr>
<td>WSens</td>
<td>( \lambda_V = S(Q_V)/\sum_{V' \in V} S(Q_{V'}) )</td>
</tr>
<tr>
<td>VSens</td>
<td>( \lambda_V = \hat{\Delta}<em>V/\sum</em>{V' \in V} \hat{\Delta}_{V'} )</td>
</tr>
</tbody>
</table>

\( Q_V \) and \( S(W) \) be the workload sensitivity of \( W \) with \( S(W) = \max_i \|w_i\|_1 \), where \( w_i \) denotes the \( i \)-th column of \( W \). Then \( W\text{-}\text{NNLS} \) computes \( \tilde{y} = y + \xi \), where each \( \xi_i \) is i.i.d. drawn from \( \text{LAPLACE}(0, \epsilon_V/S(W)) \), and returns \( \hat{x} = \arg \min_{x \geq 0} \|Wx - \tilde{y}\|_2 \).

**Dawa** We lastly use DAWA routine to estimate a vector of counts \( \tilde{x} \). DAWA partitions the vector space of \( x \) in continuous segments that have similar counts. Then it computes count estimates for the partitions, assumes uniformity within each partition, and lastly divides the noisy partition count to get estimates for the individual cells in that partition. Since we never utilize the second step of DAWA our instantiation of it is *workload agnostic*.

All three methods described are sensitive in the complexity of a view definition \( V \), as the complexity increases all methods become both intractable to use – i.e., for views with large cross-domain it might be intractable to produce the vector form \( x \). Moreover, even in the case that the size of \( V \) is not prohibitive w.r.t. the vectorization step, large view complexity leads to high cell count for \( x \) which in turn leads to high error rates of these methods.

5.5 Budget Allocator

Recall from Definition 3.2.4 that changing a row in the primary sensitive relation \( R \) results in changing \( \Delta_R(V) \) rows in view \( V \), where \( \Delta_R(V) \) is the sensitivity of view \( V \). Thus, running an \( \epsilon_V \)-DP algorithm on view \( V \) will satisfy \( (R, \Delta_R(V) \cdot \epsilon_V)\)-DP. For
that reason the any budget allocation strategy for materializing views needs to take into account the sensitivity of each view.

In PRIVSQL, budget allocation is performed by BUDGETALLOC, which has access to the intermediate non-private outputs of PRIVSQL and returns $\mathcal{E} = \{\epsilon_V\}_{V \in \mathcal{V}}$; a budget allocation that satisfies:

$$\sum_{V \in \mathcal{V}} \hat{\Delta}_V \epsilon_V \leq \epsilon',$$  \hspace{1cm} (5.8)

where $\hat{\Delta}_V$ is an upper bound of $\Delta_R(V)$ as computed from SENSALC (see Section 5.2.1) and $\epsilon'$ is the budget allocated to view generation, i.e., $\epsilon' = \epsilon - \epsilon_{mf}$. The ideal allocator would be a query fair allocator that splits the budget such that each query of the representative workload incurs the same error. In this work, we consider allocators of the following form:

$$\text{BUDGETALLOC} = \{\lambda_V \cdot \epsilon / \hat{\Delta}_V\}_{V \in \mathcal{V}}$$

As long as $\forall V \in \mathcal{V} : \lambda_V \geq 0$ and $\sum_{V \in \mathcal{V}} \lambda_V \leq 1$ this satisfies Eq. (5.8). We use 4 strategies for budget allocation as shown in Table 5.2 – NAIVE divides $\epsilon$ equally among views; WSIZE, splits the privacy budget according to the size of $Q_V$ the partial workload of each view; WSENS allocates the privacy budget according to the sensitivity of each $Q_V$; and VSENS splits the privacy budget proportionally to the sensitivity of each view.

**Naive** The first method we describe is More specifically, under NAIVE we have $\forall V \in \mathcal{V} : \lambda_V = 1/|\mathcal{V}|$. Under this naive allocation, views involving joins (with typically larger sensitivities) have lower privacy budgets and thus will support query answering with higher errors.

**Workload Size** Our next allocator is More specifically, $\forall V \in \mathcal{mathcal{V}} : \lambda_V = |Q_V| / \sum_{V' \in \mathcal{V}} |Q_{V'}|$. This allocation might be preferable in situations with highly im-
balanced partial workload sizes, where one view can be used to answer the majority of queries, while other views can only answer a handful of them.

**Workload Sensitivity** The *workload sensitivity fair* allocation strategy More specifically, $\forall V \in mathcal{V} : \lambda_V = S(Q_V) / \sum_{V' \in S(Q_{V'})}$. In the case of significant overlap between queries of a partial workload (in terms of tuples accessed), this technique is similar to WSIZE. However, it differs in the case where $|Q_V| \gg S(Q_V)$ a case that implies little to no overlap between queries of $Q_V$.

**View Sensitivity** Lastly, the *view sensitivity fair* VSENS allocation strategy splits the privacy budget proportionally to the sensitivity value of each view, with high sensitivity views receiving a higher privacy budget. More specifically, $\forall V \in \mathcal{V} : \lambda_V = \hat{\Delta}_V / \sum_{V \in \mathcal{V}} \hat{\Delta}_V$. The goal of VSENS is to permit a more uniform error among views regardless of their view sensitivity.

### 5.6 Privacy Proof

We conclude with a formal privacy statement.

**Theorem 5.6.1.** Given an acyclic schema $S = (R_1, \ldots, R_k)$ with foreign constraints $Q$ and a privacy policy $P = (\epsilon, R)$, where $R \in S$. PRIVSQL satisfies $P$-differential privacy.

*Proof.* PRIVSQL first selects and rewrites a set of views $\mathcal{V}$, then allocates the privacy budget among these views, and generates a private synopsis by executing an $\epsilon_V$-differentially private algorithm for each view $V \in \mathcal{V}$, which by Theorem 5.3.2 ensures $(R, \epsilon_V)$ differential privacy. From Eq. (5.8), BUDGETALLOC satisfies $\sum_{V \in \mathcal{V}} \hat{\Delta}(V) \cdot \epsilon_V \leq \epsilon'$. Since the budget consumed from MAXFREQCALC is $\epsilon_{mf}$ and by the sequential composition (Theorem 3.2.2), the synopsis generation phase satisfies $(R, \epsilon)$-DP, where $\epsilon = \epsilon' + \epsilon_{mf}$.

58
PRIVSQL answers queries with these private synopses without accessing the private database. By post-processing (a special case of sequential composition), the privacy guarantee $(R, \epsilon)$-DP does not change. \hfill \Box
In this chapter we focus on PrivSynGen the module responsible for releasing a single private synopsis given a fixed privacy budget. Remember that the input to PrivSynGen is a triple \((V(D), \epsilon_V, Q_V)\), where \(V(D)\) is the materialized view, \(\epsilon_V\) a privacy parameter associated with that view, and \(Q_V\) is a set of linear (to \(V\)) queries. As discussed in Section 5.4 this problem can be reduced to releasing query answers on a single table under differential privacy – a well studied problem in the literature. In the sequel we: (a) present the algorithmic landscape for releasing a synopsis of a single table; (b) describe the challenges with selecting a suitable algorithm for a given input \((V(D), \epsilon_V, Q_V)\); and (c) propose and describe Pythia, a meta-algorithm that automatically (and without additional privacy leaks) performs algorithm selection for a given input.

6.1 Background & Motivation

For the remainder, we treat the materialized view \(V(D)\) as a single relational table for which we want to answer the set of queries \(Q_V\) under \(\epsilon\)-differential privacy. The private answers to \(Q_V\) can then be used to construct the private synopsis of \(V(D)\)
as described in Section 5.4.

6.1.1 Algorithmic Landscape

For most given inputs, the algorithm with the best accuracy achievable under \( \epsilon \)-differential privacy is unknown. There are general-purpose algorithms (e.g. the Laplace Mechanism [DMNS06] and the Exponential Mechanism [MT07]), which can be adapted to a wide range of settings to achieve differential privacy. However, the naive application of these mechanisms nearly always results in sub-optimal error rates. For this reason, the design of novel differentially-private mechanisms has been an active and vibrant area of research [HLM12][LHMW14][LYQ][QYL13]-[XGX12][ZCX+14a]. Recent innovations have had dramatic results: in many application areas, new mechanisms have been developed that reduce the error by an order of magnitude or more when compared with general-purpose mechanisms and \textit{with no sacrifice in privacy}.

While these improvements in error are absolutely essential to the success of differential privacy in the real world, they have also added significant complexity to the state-of-the-art. First, there has been a proliferation of different algorithms for popular tasks. For example, in a recent survey [HMM+16], Hay et al. compared 16 different algorithms for the task of answering a set of 1- or 2-dimensional range queries on a single table. Even more important is the fact that many recent algorithms are \textit{data-dependent}, meaning that the added noise (and therefore the resulting error rates) vary between different input datasets. Of the 16 algorithms in the aforementioned study, 11 were data-dependent.

Data-dependent algorithms exploit properties of the input data to deliver lower error rates. As a side-effect, these algorithms do not have clear, analytically computable error rates, unlike their simpler data-independent counterparts. When running data-dependent algorithms on a range of different relational tables (as in the
case of the materialized views produced by PrivSQL), one may find that error is much lower for some tables, but it could also be much higher than other methods on other tables, possibly even worse than data-independent methods. The difference in error across different tables may be large, and the “right” algorithm to use depends on a large number of factors: the number of records in the table, the setting of epsilon, the domain size, and various structural properties of the data itself.

As a result, the benefits of recent research advances are unavailable in realistic scenarios. Both privacy experts and non-experts alike do not know how to choose the “correct” algorithm for privately completing a task on a given input.

6.1.2 Algorithm Selection

Motivated by this, we introduce the problem of differentially private Algorithm Selection, which informally is the problem of selecting a differentially private algorithm for a given specific input, such that the error incurred will be small.

One baseline approach to Algorithm Selection is to arbitrarily choose one differentially private algorithm (perhaps the one that appears to perform best on the inputs seen so far). We refer to this strategy as Blind Choice. As we will show later adopting blind choice does not guarantee an acceptable error for answering queries under differential privacy. A second baseline approach is to run all possible algorithms on the sensitive database and choose the best algorithm based on their error, we refer to this strategy as Informed Decision. This approach, while seemingly natural, leads to a privacy leak since checking the error of a differentially private algorithm requires access to the sensitive data.

6.1.3 Our approach

We propose Pythia, an end-to-end differentially private mechanism for achieving near-optimal error rates using a suite of available privacy algorithms. Pythia is a
Figure 6.1: The Pythia meta-algorithm computes private query answers given the input data, workload, and epsilon. Internally, it models the performance of a set of algorithms, automatically selects one of them, and executes it.

meta-algorithm, which safely performs automated Algorithm Selection and executes the selected algorithm to return a differentially private result. Using Pythia, data curators do not have to understand available algorithms, or analyze subtle properties of their input data, but can nevertheless enjoy reduced error rates that may be possible for their inputs.

Pythia works in three steps, as illustrated in Fig. 6.1. First it privately extracts a set of feature values from the given input. Then, using a Feature-based Algorithm Selector Pythia chooses a differentially private algorithm $A^*$ from a collection of available algorithms. Lastly, it runs $A^*$ on the given input. An important aspect of this approach is that Pythia does not require intimate knowledge of the algorithms from which it chooses, treating each like a black-box. This makes Pythia extensible, easily accommodating new advances from the research community as they appear.

Our results have two important consequences. First, because our Feature-based Algorithm Selector is interpretable, the output of training phase can provide insight into the space of algorithms and when they work best. (See for example Fig. 6.3).
Second, we believe our approach can have a significant impact on future research efforts. An extensible meta-algorithm, which can efficiently select among algorithms, shifts the focus of research from generic mechanisms (which must work well across a broad range of inputs) to mechanisms that are specialized to more narrow cases (e.g., datasets with specific properties). One might argue that algorithms have begun to specialize already; if so, then effective meta-algorithms justify this specialization and encourage further improvements. In this section we describe the data model, workloads, differentially private algorithms, and our error metric.

**Data Model:** We use the vector representation shown in Section 2.2.1. As a reminder, the relational table $D$ is a multiset of records, each having $k$ attributes with discrete and ordered domains. We describe $D$ as a vector $x \in \mathbb{N}^d$ where $d = d_1 \times \ldots \times d_k$, and $d_j$ is the domain size of the $j^{th}$ attribute. We denote the $i^{th}$ value of $x$ with $x_i$.

Given a vector dataset $x$, we define three of its key properties: its *scale* is the total number of records: $s_x = \|x\|_1$; its *shape* is the empirical distribution of the data: $p_x = x/s_x$; and its *domain size* is the number of entries $d_x = |x|$.

**Queries** A *query workload* is a set of queries defined on $x$ and we use matrix notation to define it. A query workload $W$ is an $m \times d$ matrix where each row represents a different linear query on $x$. The answer to this workload is defined as $y = Wx$. An example of a workload is $P$, an upper triangular matrix with its non-zero elements equal to 1. This workload is called the *prefix workload* and contains all prefix queries on a dataset vector – i.e., $\forall i : q_i = x_1 + \ldots + x_i$.

Usually a data curator is not interested in answering one specific workload, but rather a collection of similar workloads. For that reason we define a *task* $T$ as a collection of relevant workloads. Examples of tasks include 1D range queries, 2D range queries, marginal queries, etc.
Table 6.1: Algorithm overview for query release on single table.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Tasks</th>
<th>Prior Work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Independent</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Laplace</td>
<td>General Purpose</td>
<td>[DMNS06]</td>
</tr>
<tr>
<td>Hb</td>
<td>Range Queries</td>
<td>[QYL13]</td>
</tr>
<tr>
<td>Privelet</td>
<td>Range Queries</td>
<td>[XWG11]</td>
</tr>
<tr>
<td><strong>Data Dependent</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIFORM</td>
<td>General Purpose</td>
<td>n/a</td>
</tr>
<tr>
<td>DAWA</td>
<td>Range Queries</td>
<td>[LHMW14]</td>
</tr>
<tr>
<td>MWEM</td>
<td>General Purpose</td>
<td>[HLM12]</td>
</tr>
<tr>
<td>AHP</td>
<td>General Purpose</td>
<td>[ZCX+14a]</td>
</tr>
<tr>
<td>AGRID</td>
<td>2d Range Queries</td>
<td>[LYQ]</td>
</tr>
<tr>
<td>DPCUBE</td>
<td>2d Range Queries</td>
<td>[XGX12]</td>
</tr>
</tbody>
</table>

**Error Measurement** For a differentially private algorithm $A$, dataset $x$, workload $W$, and privacy parameter $\epsilon$ we denote the output of $A$ as $\tilde{y} = A(W, x, \epsilon)$ . Then the error is the $L_2$ distance between the vectors of the true answers and the noisy estimates: $\text{error}(A, W, x, \epsilon) = \|\tilde{y} - y\|_2$

**Algorithms** Differentially private algorithms can be broadly classified as data-independent and data-dependent algorithms. The error introduced by data independent algorithms is independent of the input database instance. Classic mechanisms like the Laplace mechanism [DMNS06] are data independent. For the task of answering range queries, alternative data-independent techniques can offer lower error. One example is Hb [QYL13], which is based on hierarchical aggregation – i.e., it computes counts for both individual bins of a histogram as well as aggregate counts of hierarchical subsets of the bins.

Data-dependent algorithms usually spend a portion of the budget to learn a property of the dataset based on which they calibrate the noise added to the counts of $x$. A category of data-dependent algorithms are partition-based; these algorithms work by learning a partitioning of $x$ and add noise only to the aggregate counts of the
partitions. The value of any individual cell of \( x \) is given by assuming uniformity on its partition. While this technique reduces the total noise added to \( x \), it also introduces a bias factor because of the uniformity assumption on the partitions. Hence, the overall error greatly depends on the shape of \( x \). Examples of data-dependent partitioning algorithms include DAWA, AGRID, AHP, and DPCUBE. Other data-dependent algorithms (like MWEM) use other data adaptive strategies.

Section 6.1.3 lists the algorithms that Pythia chooses from for answering the task of 1- and 2-dimensional range queries.

6.2 Algorithm Selection

In this section we formally define the problem of Algorithm Selection, describe the desiderata of potential solutions, and discuss the limitations of three baseline approaches.

**Example 5. Histogram Release** Suppose a medical establishment that wants to share aggregate statistics of their medical records to teams of researchers. More specifically, the medical researchers have requested a histogram of illnesses of all the patients in the last year. The hospital wants to honor this request, while being careful about any privacy leaks that such a release might have. For that reason they want to release a differentially private version of the histogram over the diseases. The data curator of the hospital has a basic understanding of histogram release under differential privacy and is familiar with the literature. However and since the curator is not a privacy expert, he has no good insight on what is the best in terms of error algorithms to choose from.

**Example 6. Multi-stage Task** Suppose that a credit card company wants to offer a new product to its clients, a credit default estimator that warns a client if their current behavior might lead to a future credit default. The way that such a service works is
simple, based on historical data of their older clients, the credit card company builds a binary classifier that it is then used by new users to estimate their probabilities of defaulting. The input of such a model are histograms of attributes of the data like income, past payments, demographics, etc, of past clients. Since the credit card company does not want to compromise the privacy of their older clients, they decide to use differential privacy to estimate the histograms before building the binary classifier. If the credit card company uses only one algorithm to estimate all the histograms needed, they miss on a big opportunity to improve the accuracy of the classifier, since the set of these histograms is highly heterogeneous.

We identify two important properties of modern differentially private applications, algorithm suitability and input heterogeneity, both of which motivate algorithm selection. In Example 5 we show a use case that highlights the importance of algorithm suitability, the data curator wants to use the algorithm that provides the highest utility. In Example 6 we illustrate that the increasing complexity of modern differentially private systems leads to input heterogeneity, i.e., the sensitive data is not a single histogram, but rather a collection of histograms. To address both of these limitations, we propose using algorithm selection to choose a differentially private algorithm before answering queries.

Algorithm selection is function over a suite of algorithms, a sensitive dataset, a workload of queries for the dataset, and a desired privacy loss budget associated with the query release. More specifically,

**Definition 6.2.1. Algorithm Selection.** Let $W$ be a workload of queries to be answered on database $x$ under $\epsilon$-differential privacy. Let $A$ denote a set of differentially private algorithms that can be used to answer $W$ on $x$. The problem is to select an algorithm $A^* \in A$ to answer $W$ on $x$.

We identify the following desiderata for Algorithm Selection solutions: (a) dif-
ferentially private, (b) algorithm-agnostic, and (c) competitive.

**Differentially Private:** Algorithm Selection methods must be differentially private. If the input data is relevant to an Algorithm Selection method, any use of the input data must be included in an end-to-end guarantee of privacy.

**Agnostic:** Algorithm Selection methods should treat each algorithm \( A \in \mathcal{A} \) as a black box, i.e., solutions should only require that algorithms satisfy differential privacy and should be agnostic to the rest of the details of each algorithm. Agnostic methods are easier to deploy and are also readily extensible as research provides new algorithmic techniques.

**Competitive:** Algorithm Selection methods should provide an algorithm \( A^* \) that offers low error rates on a wide variety of inputs (multiple workloads, different datasets).

We measure the competitiveness of an Algorithm Selection method using a *regret* measure defined to be the ratio of the error of the selected algorithm to the least error achievable from any algorithm of \( \mathcal{A} \). More precisely, given a set of differentially private algorithms \( \mathcal{A} \), a workload \( W \), a dataset \( x \), and a privacy budget \( \epsilon \), we define the (relative) regret with respect to \( \mathcal{A} \), of an algorithm \( A \in \mathcal{A} \) as follows:

\[
\text{regret}(A, W, x, \epsilon) = \frac{\text{error}(A, W, x, \epsilon)}{\min_{A \in \mathcal{A}} \text{error}(A, W, x, \epsilon)}
\]

### 6.2.1 Baseline Approaches

As we mentioned in Section 6.1, two baseline approaches to Algorithm Selection are *Blind Choice* and *Informed Decision*. We also consider a third baseline, *Private Informed Decision* and explain how each of these approaches violate our desiderata.

**Blind Choice** This baseline consists of simply selecting an arbitrary differentially private algorithm and using it for all inputs. It is a simple solution to Algorithm
Selection and clearly differentially private. But such an approach will only be competitive if there is one algorithm that offers minimal, or near-minimal error, on all inputs. Hay et al. demonstrated [HMM+16] that the performance of algorithms varies significantly across different parameters of the input datasets, like domain size, shape, and scale. One of the main findings is that there is no single algorithm that dominates in all cases. Our results in Section 7.2.2 confirm this, showing that the regret of Blind Choice (for any one algorithm in $\mathcal{A}$) is high.

**Informed Decision** In *Informed Decision* the data curator first runs all available algorithms on the given input and records the error of each algorithm. He then chooses the algorithm that performed the best. While Informed Decision solves Algorithm Selection with the lowest possible regret, it violates differential privacy since it needs to access the true answers in order to compute the error.

**Theorem 6.2.1.** There exists a set of differentially private algorithms $\mathcal{A}$, an input $(W, x, \epsilon)$ such that if Informed Decision is used to choose $A^* \in \mathcal{A}$ for the input $(W, x, \epsilon)$ then releasing $A^*(W, x, \epsilon)$ violates Differential Privacy.

**Proof.** Let $W$ be a query workload and let $x$ and $y$ be two neighboring datasets (i.e., $\|x - y\|_1 = 1$) that have distinct outputs on $W$. That is, $Wx \neq Wy$. Let $A_x$ and $A_y$ be two algorithms such that $A_x$ always outputs $Wx$ independent of the input, and $A_y$ always outputs $Wy$ independent of the input. Since $A_x$, and $A_y$ are constant functions, they trivially satisfy differential privacy for any $\epsilon$ value.

Consider the Algorithm Selection problem where $\mathcal{A} = \{A_x, A_y\}$. For input $x = (W, x, \epsilon)$ informed decision picks the algorithm that results in the least error which is $A_x$. For informed decision $ID$ to satisfy $\epsilon$-differential privacy, we want $\forall S \in \text{Range}(ID)$:

$$P(\text{ID}(x) \in S) \leq \exp(\epsilon) \times P(\text{ID}(y) \in S)$$
But we know that $P(\text{ID}(x) = Wx) = 1$, while $P(\text{ID}(y) = Wx) = 0$, resulting in contradiction.

**Private Informed Decision** This strategy follows the same steps as Informed Decision except that estimation of the error of each algorithm is done in a differentially private manner. Naturally, this means the total privacy budget must be split to be used in two phases: (a) private algorithm error estimation, and (b) running the chosen algorithm. This kind of approach has already been proposed in [CV13], where the authors use this method to choose between differentially private machine learning models.

The main challenge with this approach is that it requires that algorithm error has low sensitivity; i.e., adding or removing a record does not significantly impact algorithm error. However, we are not aware of tight bounds on the sensitivity of error for many of the algorithms we consider in Section 7.1.2. This means that Private Informed Decision cannot be easily extended with new algorithms. So, while Private Informed Decision satisfies differential privacy and may be more competitive than Blind Choice, it violates the algorithm agnostic desideratum.

### 6.3 Pythia Overview

Our approach to solve Algorithm Selection is called *Pythia* (see Fig. 6.1) and works as follows. Given an input $(W, x, \epsilon)$, Pythia first extracts a set of features $F$ from the input, and perturbs each $f \in F$ by adding noise drawn from $\text{LAPLACE}(d \cdot \Delta f / \epsilon_1)$, where $\Delta f$ denotes the sensitivity of $f$, and $d$ is the number of sensitive features. The set of features and their sensitivities are predetermined. Next it uses a Feature-based Algorithm Selector (FAS) to select an algorithm $A^*$ from an input library of algorithms $\mathcal{A}$ based on the noisy features of the input. Finally, Pythia executes algorithm $A^*$ on $(W, x, \epsilon_2)$ and outputs the result. It is easy to see that this process
is differentially private.

**Theorem 6.3.1.** *Pythia satisfies* $\epsilon$*-Differential Privacy, where* $\epsilon = \epsilon_1 + \epsilon_2$.

*Proof.* Feature extraction satisfies $\epsilon_1$-Differential Privacy and executing the chosen algorithm satisfies $\epsilon_2$-Differential Privacy. The proof follows from sequential composition of differential privacy (see Theorem 2.1.1).

The key novelty of our solution is that the Feature-based Algorithm Selector is constructed using a learning based approach, called Delphi (see Fig. 6.2). Delphi can be thought of as a constructor to Pythia: given a user-specified task $T$ (e.g., answering 1-dimensional range queries) it utilizes a set of differentially private algorithms $\mathcal{A}_T$ that can be used to complete the task $T$, and a set of public datasets to output the set of features $F$, their sensitivities $\Delta F$ as well as the Feature-based Algorithm Selector (FAS). To learn the FAS, Delphi constructs a training set by (a) generating training inputs $(\mathbf{W}, \mathbf{x}, \epsilon)$ that span diverse datasets and workloads, and (b) measuring the empirical error of algorithms in $\mathcal{A}_T$ on training inputs. Delphi never accesses the private input database instance, but rather uses public datasets to train the FAS.
This allows Delphi to (a) trivially satisfy differential privacy with $\epsilon = 0$, and (b) be run once and re-used for Algorithm Selection on different input instances.

Next we describe the design of Delphi and Pythia in detail. Section 6.4 describes the training procedure employed by Delphi to learn a Feature-based Algorithm Selector. Section 6.4.2 describes specific implementation choices for the task of answering range queries. Section 6.5 describes the Pythia algorithm as well as optimizations that help reduce error.

6.4 Delphi: Learning a FAS

Delphi's main goal is to build a Feature-based Algorithm Selector (FAS) that can be used by Pythia for algorithm selection. The design of Delphi is based on the following key ideas:

**Data Independent** As mentioned in the previous section, we designed Delphi to work without knowledge of the actual workload $W$, database instance $x$, or privacy parameter $\epsilon$ that will be input to Pythia. Delphi only takes the task (e.g., answering range queries in 1D) as input. First, this saves privacy budget that can be used for extracting features and running the chosen algorithm later on. Secondly, this allows the FAS output by Delphi to be reused for many applications of the same task.

**Rule Based Selector** The FAS output by Delphi uses rules to determine how features are mapped to selected algorithms. In particular we use Decision Trees [Loh11] for algorithm selection. Decision trees can be interpreted as a set of rules that partition the space of inputs (in our case $(W, x, \epsilon)$ triples), and the trees Delphi outputs shed light into the classes of $(W, x, \epsilon)$ for which an algorithm has the least error. Moreover, prediction is done efficiently by traversing the tree from root to leaf. We discuss our decision tree implementation of FAS in Section 6.4.

**Supervised Approach** Delphi constructs a training set where each training instance
is associated with features extracted from triples \((W, x, \epsilon)\) and the empirical error incurred by each \(A \in \mathcal{A}\) for that triple. We ensure the training instances captures a diverse set of \(\epsilon\) values as well as databases \(x\) with varying shapes, scales and domain sizes. Unlike standard supervised learning where training sets are collected, Delphi can (synthetically) generate as many or as few training examples as necessary. Training set construction is explained in Section 6.4.

**Regret-based Learning** Standard decision tree learning assumes each training instance has a set of features and a label with the goal of accurately predicting the label using the features. This can be achieved by associating each training instance with the algorithm achieving the least error on the instance. However, standard decision tree algorithms view all mispredictions as equally bad. In our context this is not always the case. Recent work [HMM+16] has shown that for datasets \(x\) with large scales (e.g. \(\geq 10^8\) records), algorithms like MWEM have a high regret (in the hundreds), while algorithms like Hb and DAWA have low regrets (close to 2) for the task of 1D range queries. A misprediction that offers a competitive regret should not have the same penalty as a misprediction whose regret is in the hundreds. Towards this goal, Delphi builds a decision tree that partitions the space of \((W, x, \epsilon)\) triples into regions where the average regret attained by some algorithm is low. Delphi does not distinguish between algorithms with similar regrets (since these would all be good choices), and thus is able to learn a FAS that selects algorithms with lower regret than models output by standard decision tree learning. Our learning approach is described in detail in Section 6.4.1.

We use decision trees to implement the Feature-based Algorithm Selector. The FAS is a binary tree where the internal nodes of the tree are labeled with a feature and a condition of the form \(f_i \leq v\). Leaves of the tree determine the outcome, which in our case is the chosen algorithm. The decision tree divides the space of inputs
into non-overlapping regions – one per leaf. All inputs in the region corresponding to the leaf satisfy a conjunction of constraints on features $\ell_1 \land \ell_2 \land \ldots \land \ell_h$, where $\ell_i = (f_i \leq v)$ if the leaf is in the left sub-tree of an internal node with that condition, and $\ell_i = (f_i > v)$ if the leaf is in the right sub-tree.

Given an unseen input set of features, prediction starts at the root of the tree. The condition on the internal node is checked. Traversal continues to the left child if the condition is true and to the right if the condition is false. Traversal stops at the leaf which determines the outcome. Figure 6.3 shows an example FAS for the task of 2-dimensional range queries. For instance, the FAS selects the LAPLACE mechanism for inputs with small domain size ($\leq 24$) but a large number of records ($> 3072$). Similarly, the FAS picks AGRID for large domain sizes ($> 24$) with a small number of non-zero (NNZ $\leq 25$) counts.

**Training Data** For a task $T$, Delphi chooses a set of differentially private algorithms $\mathcal{A}_T$ for $T$. Then using a library of representative workloads for the task $T$ and a benchmark of public datasets, Delphi constructs a set of inputs $\mathcal{Z}_T$ of the form $z = (\mathbf{W}, \mathbf{x}, \epsilon)$. Details on how $\mathcal{Z}_T$ is constructed can be task dependent, and the implementation for range queries is described in Section 6.4.2.
Algorithm 4 CART ($\mathcal{I}$) [BFOS84, Loh11]

1: Start at the root node, containing all training data $\mathcal{I}$.
2: For each feature $f$ find the value $s^*$ such that splitting on $(f, s^*)$ results in children whose weighted average of node impurity (NI) is minimized. Repeat the process for all features and choose $(f^*, s^*)$ that minimizes the weighted average of NI of the children.
3: Recurse on each child until the stopping criterion is met.

Next, from an input $z = (W, x, \epsilon)$, we extract a feature vector to be used in FAS. Features can be derived from the workload $W$, the input dataset $x$, or the privacy budget $\epsilon$. Let $F$ be a set of real valued functions over input triples. For $f \in F$, we denote by $f_z$ the value of feature $f$ on input triple $z$, and by $f_z$ the feature vector $[f_{1z}, \ldots, f_{mz}]^T$. Examples of features include the number of records in the dataset (or scale), or the domain size. Section 6.4.2 describes the precise set of features used for the task of range queries. Delphi also records the performance of each algorithm $A \in \mathcal{A}_T$ on each input $z \in \mathcal{Z}_T$ and creates a regret vector for each $z$: $r_z$ that contains the regret for all algorithms in $\mathcal{A}_T$ for input $z$.

$$r_z = [\text{regret}_{rel}(A, z)]^T_{A \in \mathcal{A}_T}$$

Finally, Delphi records the algorithm with the least error on $z$, say $A_z^*$, which will have a regret of 1. Thus, the final training data is a set $\mathcal{I}$ consisting of triples of the form $i = (f_z, A_z^*, r_z)$. We use the notation $i.f_z, i.A_z^*, i.r_z$ to refer to the different members of the training instance $i$.

6.4.1 Regret-based Learning

Decision trees are typically constructed in a top-down recursive manner by partitioning the training instances $\mathcal{I}$ into a tree structure. The root node is associated with the set of all training examples. An internal node $v$ that is associated with a subset of training examples $V \subset \mathcal{I}$, is split into two child nodes $v_{f \leq s}$ and $v_{f > s}$ based on a condition $f \leq s$. The children are associated with $V_{f \leq s} = \{i \in V | i.f_z \leq s\}$ and $V_{f > s} = \{i \in V | i.f_z > s\}$, respectively. The split condition $f \leq s$ is chosen by
computing the values $f^*, s^*$ according to a splitting criterion. Recursive tree construction ends when a stopping condition is met. The two conditions we consider are: (a) when no split of the node $v$ results in an improvement and (b) when the tree has reached a maximum depth $h_{\text{max}}$. Algorithm 4 describes a standard decision tree construction algorithm called CART. Note that the computation of $f^*$ implies that features are automatically selected in order from the system.

The splitting criterion we use in this work chooses $(f^*, s^*)$ to maximize the difference between the node impurity (NI for short) of the parent node, and the weighted average of the node impurities of the children resulting from a split.

$$\text{argmax}_{f, s} \left( |V|\text{NI}(v) - (|V_{f \leq s}|\text{NI}(v_{f \leq s}) + |V_{f > s}|\text{NI}(v_{f > s})) \right)$$

Node impurity NI is a function that maps a set of training instances to a real number in the range $[0, 1]$ and measures the homogeneity of the training examples within a node with respect to predicted values. In our context, NI($v$) should be low if a single algorithm achieves significantly lower error than all other algorithms on instances in $V$, and high if many algorithms achieve significantly lower error on subsets training examples. Decision tree construction methods differ in the implementation of NI.

We next describe four alternate implementations of NI that result in four splitting criteria – best algorithm, group regret, minimum average regret and regret variance criterion. As the names suggest, the first criterion is based just on the best algorithm for each training instance (and is an adaptation of a standard splitting criterion). The other three splitting criteria are novel and are based on the regrets achieved by all algorithms in $A_T$ on a training instance $z$. In Section 7.2.4 we make a quantitative comparison between all splitting criteria we consider.

**Best Algorithm Criterion** This approach treats the problem of Algorithm Selection as a standard classification problem, where each training instance is associated
with a label corresponding to the algorithm with the least error on that instance. If multiple algorithms achieve a regret of 1, one of them is arbitrarily chosen. The NI(V) implementation we consider is the standard Gini impurity [Loh11], which measures the likelihood that a randomly chosen training instance in V will be misclassified if a label was predicted based on the empirical distribution of labels in V. More specifically, for node v of the tree let \( t_v \) denote the empirical distribution over the labels.

\[
t_v = \left[ \frac{1}{|V|} \left| \{ i \in V | s.t. \ i.A_i^* = A \} \right| \right]_\forall A \in \mathcal{A}
\]

That is, \( t_v[A] \) is the fraction of training instances for which A is the best algorithm. The Gini impurity on node v is defined as follows:

\[
\text{NI}(v) = \text{Gini}(v) = 1 - t_v^\top \cdot t_v
\]

As discussed before, the best algorithm criterion views all algorithms that are not the best as equally bad. Delphi employs a \textit{regret-based} splitting criterion discussed next, which allow to rank different splits based on their average regret. Recall that \( i.r_z \) denotes the vector of regrets for all algorithms \( A \in \mathcal{A} \) on training instance \( z \).

We define the average regret vector of training instances in V as:

\[
r_v = \frac{1}{|V|} \sum_{i \in V} i.r_z
\]

**Group Regret Criterion** We now present our best splitting criterion for algorithm selection, which we call the Group Regret Criterion. The key idea behind this splitting criterion is to (a) cluster algorithms with similar average regrets for a set of training instances, (b) associate training instances of a node v to the group of v with the least average regret, and (c) compute the Gini impurity criterion on the
empirical distribution of the groups rather than on the empirical distribution over
the labels (i.e., the best algorithm). The intuition is that choosing any algorithm
from the same cluster would result in similar average regret, and thus algorithms in
a cluster are indistinguishable.

Let $C$ a partitioning of $A_T$, then for a node $v$ let $g_{vc}$ denote the empirical distri-
bution over the clusters of $C$:

$$g_{vc} = \left[ \frac{1}{|V|} \{ i \in V \mid s.t. \ i.A^*_z \in C \} \right]_{\forall C \in C}^T$$

That is, $g_{vc}[C]$ is the fraction of training instances for which some $A \in C$ is the
algorithm that attains the least error.

**Definition 6.4.1 (θ-Group Impurity).** Given a node $v$ associated with a set of train-
ing examples $V$ and a threshold $\theta \in \mathbb{R}^+$, we define a $\theta$-clustering of algorithms
$A_T$ to be a partitioning $C = \{ C_1, \ldots, C_k \}$ such that $\forall C \in C$ and $\forall A, A' \in C$,
$|r_v[A] - r_v[A']| \leq \theta$. The $\theta$-Group Impurity of $v$ is defined as:

$$\text{NI}(v) = \text{GI}_\theta(v) = \min_{\theta\text{-clusterings } C} 1 - g_{vc}^T \cdot g_{vc}$$ (6.1)

For a node $v$, the clustering $C^*$ that achieves the minimum $\text{GI}_\theta(v)$ is called the
$\theta$-Group Clustering ($\theta\text{GC}$).

The intuition behind $\theta$-Group Impurity is the following: suppose $A$ is the best
algorithm for an instance $z$ (regret is 1). Other algorithms $A'$ that are in the same
cluster in a $\theta\text{GC}$ have regret at most $\theta + 1$, and hence the model should not be
penalized for selecting $A'$ instead of $A$. However, the FAS must be penalized for
selecting algorithms that are not in the same cluster as $A$ in the $\theta\text{GC}$.

$\theta$-group clusterings can be efficiently computed due to the following property:

**Lemma 6.4.1.** Let $C$ be a $\theta\text{GC}$ for a set of algorithms in node $v$ of the FAS. For
any three algorithms $K, L, M$ such that $r_v[K] \leq r_v[L] \leq r_v[M]$, if $K$ and $L$ are in
the same cluster \( C \in \mathcal{C} \), then \( L \) is also in the same cluster \( C \). Any \( \theta \mathrm{GC} \): \( \mathcal{C}^* \) is regret-continuous. For \( C \in \mathcal{C}^* \) and any three algorithms \( K, L, M \in \mathcal{A}_T \) such that 
\[
\mathbf{r}_v[K] \leq \mathbf{r}_v[L] \leq \mathbf{r}_v[M]
\] if \( K, M \in C \) then \( L \in C \).

Before we prove Lemma 6.4.1, we extend our notation to help us with the proof. Let a \( \theta \)-clustering \( \mathcal{C} \), then the partial sum of a cluster \( C_i \in \mathcal{C} \) is: 
\[
S_i = g_{\mathcal{C}}[C_i] \mathbf{g}_{\mathcal{C}}[C_i],
\]
it follows that 
\[
\mathbf{g}_{\mathcal{C}}^T \cdot \mathbf{g}_{\mathcal{C}} = \sum_{C_i \in \mathcal{C}} S_i.
\]
Also let 
\[
g(C) = \mathbf{g}_{\mathcal{C}}^T \cdot \mathbf{g}_{\mathcal{C}}.
\]

Proof of Lemma 6.4.1. We prove by contradiction. Let \( \mathcal{C}^* \) a \( \theta \)-Group Clustering for node \( v \) and algorithms \( \mathcal{A}_T \). This implies that 
\[
\mathcal{C}^* = \arg \max_{\mathcal{C}} g(\mathcal{C}).
\]
Assume that \( \mathcal{C}^* \) does not satisfy the claim, i.e., there exist algorithms \( K, L, M \in \mathcal{A}_T \) such that 
\[
\mathbf{r}_v[K] \leq \mathbf{r}_v[L] \leq \mathbf{r}_v[M]
\] with \( K, M \in C_i \) and \( L \in C_j \), where \( C_i, C_j \in \mathcal{C}^* \). It is obvious that \( L \) is admissible to \( C_i \) (since it is bounded by \( K \) and \( M \) already in \( C_i \)).

Also note that since 
\[
\max_{h \in C_j} |\mathbf{r}_v[h] - \mathbf{r}_v[L]| \leq \theta,
\]
at least one of \( K, M \) is admissible to \( C_j \).

We consider two cases, regarding the partial sums of \( C_i \) and \( C_j \). If \( S_i^* \geq S_j^* \): we construct another solution \( \mathcal{C}' \) by removing \( L \) from \( C_j \) and adding it to \( C_i \), i.e. 
\[
\mathcal{C}' = \{ C \mid \forall C \in \mathcal{C} \setminus \{C_i, C_j\} \} \cup \{ C_j \setminus \{L\} \} \cup \{ C_i \cup \{L\} \}.
\]
The value of this solution is computed as follows:
\[
g(\mathcal{C}') = g(\mathcal{C}^*) - S_i^* - S_j^* + S_i^* - S_j^* + 2 \mathbf{t}_v[L]^2 - 2 \mathbf{t}_v[L]S_j^* + 2 \mathbf{t}_v[L]S_i^*
\]
\[
= g(\mathcal{C}^*) + 2 \mathbf{t}_v[L]^2 + 2 \mathbf{t}_v[L](S_i^* - S_j^*) \geq g(\mathcal{C}^*)
\]

If \( S_i^* \leq S_j^* \): w.l.o.g. assume only \( K \) is admissible to \( C_j \), then we construct \( \mathcal{C}' \) by removing \( K \) from \( C_i \) and adding it to \( C_j \), i.e. 
\[
\mathcal{C}' = \{ C \mid \forall C \in \mathcal{C} \setminus \{C_i, C_j\} \} \cup \{ C_j \cup \{K\} \} \cup \{ \}.
\]
\{K\} \cup \{C_i \setminus \{K\}\}. The value of this solution is computed as follows:

\[
g(C') = g(C^*) - S_i^* - S_j^* + S_i^{\prime*} + S_j^{\prime*}
\]

\[
= g(C^*) - S_i^* - S_j^* + (S_i^* - t_v[K])^2 + (S_j^* + t_v[K])^2
\]

\[
= g(C^*) + 2t_v[K]^2 + 2t_v[K]S_j^* - 2S_i^* - S_i^{\prime*}
\]

\[
= g(C^*) + 2t_v[K]^2 + 2t_v[K](S_j^* - S_i^{\prime*}) \geq g(C^*)
\]

As a consequence of Lemma 6.4.1, if the algorithms in \(A_T\) are sorted in increasing order of their regrets, then the \(\theta GC\) always corresponds to a range partitioning of the sorted list of algorithms. More precisely, if \(\{A_1, A_2, \ldots \}\) are such that \(r_v[A_i] \leq r_v[A_j]\) for all \(i \leq j\), then every cluster \(C \in C^*\) is a range \([k, m]\) such that \(\forall \ell \in [k, m]: A_\ell \in C\). When the cardinality of \(A_T\) is low (like in our experiments) one can enumerate over all the range partitions of the sorted list of algorithms to find the \(\theta GC\). In cases where \(A_T\) is large we can use dynamic programming (like in [JKM+98]) since the optimization criterion (Equation 6.1) satisfies the optimal substructure property.

**Minimum Average Regret Criterion** With minimum average regret (MAR) criterion our goal is to promote splits in the tree where the resulting average regret of the children is less than the average regret of the parent node. This is achieved by choosing a NODE-IMPURITY that measures the average regret of the node:

\[
NI(v) = MAR(v) = \frac{\|r_v\|_1}{|A_T|}
\]

**Regret Variance Criterion** The next criterion we consider is to promote splits where the variance of the regret vectors of the children is smaller than the variance of the regret of the parent node. In this case NODE-IMPURITY\((v)\) is simply the variance of \(v\):

\[
NI(v) = VAR(v) = \frac{1}{|A_T|} \sum_{A \in A_T} (r_v[A] - \frac{\|r_v\|_1}{|A_T|})^2
\]
6.4.2 Delphi for Range Queries

In this section we present how Delphi generates the set of input instances \( Z_T = \{(W, x, \epsilon)\} \) for tasks of range queries. Section 6.4.2 details how we generate \( x \)'s, and Sections 6.4.2 and 6.4.2 explain how we handle workloads and epsilon values in the training phase.

Generating Datasets

Recent work [HMM+16] on the empirical evaluation of differentially private algorithms for answering range queries identified that algorithm error critically depends on three parameters of a dataset \( x \): scale, shape, and domain size. The characteristics of the input to Pythia are not known a priori, thus we must ensure that Delphi creates training data that spans a diverse range of scales, shapes, and domain sizes.

Delphi starts with a benchmark of public datasets \( D_{public} \). One or two dimensional datasets are constructed by choosing one or two attributes from the dataset, respectively. For each choice of attribute(s), if the domain is categorical it is made continuous using kernel density estimation. This process results in an empirical density, which we call the shape \( p \). We denote by \( P \) the set of all shapes constructed.

Next, the continuous domain is discretized using equiwidth bins (in 1- or 2-dimensions) to get various domain sizes. We denote by \( K \) the set of domain sizes for each shape. Finally, to get a dataset of scale \( s \), given a domain size \( k \) and shape \( p \), we scale up the shape \( p \) by \( s \) to get a total histogram count of \( s \). The set of scales generated is denoted by \( S \). Thus the space of all datasets corresponds to \( P \times K \times S \). We denote by \( X \) the resulting set of datasets.

Workload Optimization

Replicating training examples for every possible workload for a given task would make training inefficient. Hence, we use the following optimization. Delphi maps each task
$T$ to a set of representative workloads $W_T$, which contains workloads relevant to the task. For example if $T$ is "Answer range queries on 1D datasets", then $W_T$ contains $I$ and $P$, the identity and prefix workloads respectively. The identity workload is effective as answering short range queries, while the prefix workload is a better choice for answering longer random range queries. Given a new task $T$, Delphi selects a set of differentially private algorithms $A_T$, a set of representative workloads $W_T$, and a privacy budget $\epsilon$. Delphi also generates a set of input datasets $\mathcal{X}$ (as described above).

For every workload $W \in W_t$ Delphi generates a set of training instances $I_W$ by running all algorithms of $A_T$, for all datasets $x \in \mathcal{X}$, workload $W$, and privacy budget $\epsilon$. Then Delphi uses the CART algorithm with training data $I_W$ and creates a set of FAS’s: $\{\text{FAS}_W \mid \forall W \in W_T\}$. Lastly, Delphi creates a root $r$ connecting each $\text{FAS}_W$ where edges incident to $r$ have rules based on workload features. The resulting tree with root $r$ is the FAS returned by Delphi.

**Privacy Budget Optimization**

As with workloads, Delphi could train different trees for different $\epsilon$ values. However, this would either require knowing $\epsilon$ (or a range of $\epsilon$ values) up front, or would require building an infinite number of trees. Delphi overcomes this challenge by learning a FAS for a single value of $\epsilon = 1.0$; i.e., all training instances have the same value of $\epsilon$. At run-time in Pythia, if $z = (W, x, \epsilon')$, where $\epsilon' \neq \epsilon$, Pythia transforms the input database $x$ to a different database $x' = \epsilon' x$, and runs algorithm selection on $z' = (W, x', \epsilon)$. This strategy is justified due to the scale-epsilon exchangeability property defined below.

**Definition 6.4.2.** Scale-epsilon exchangeability [HMM+16] Let $p$ be a shape, $W$ a workload. For datasets $x_1 = s_1p$ and $x_2 = s_2p$, a differentially private algorithm $A$ is scale-epsilon exchangeable if

$$\text{error}(A, W, x_1, \epsilon_1) = \text{error}(A, W, x_2, \epsilon_2)$$

whenever
$\epsilon_1 s_1 = \epsilon_2 s_2$.

Recent work [HMM+16] showed that all state-of-the-art algorithms for answering range queries under differential privacy satisfy scale-epsilon exchangeability. We can show that under asymptotic conditions, the algorithm selected by a FAS on $(W, x, \epsilon')$ that is trained on input instances with privacy parameter $\epsilon'$ would be identical to algorithm selected by a FAS' on $(W, \epsilon' x, \epsilon)$ trained on input instances with privacy parameter $\epsilon$.

Let $X$ be $P \times K \times \mathbb{R}^+$ a set of datasets. We construct inputs $Z_1 = \{(W, x, \epsilon_1) | \forall x \in X\}$ and $Z_2 = \{(W, x, \epsilon_2) | \forall x \in X\}$. We construct $I_1$ and $I_2$ by executing epsilon-scale exchangeable algorithms $A$, on $Z_1$ and $Z_2$ respectively. Let the Feature-based Algorithm Selectors constructed from these training datasets: $FAS_1 = \text{Cart}(I_1)$, and $FAS_2 = \text{Cart}(I_2)$.

**Theorem 6.4.1.** Consider instances $z_1 = (W, x_1, \epsilon_1)$ and $z_2 = (W, x_2, \epsilon_2)$ such that $\epsilon_1 x_1 = \epsilon_2 x_2$. During prediction, let the traversal of $z_1$ on $FAS_1$ result in leaf node $v_1$, and let the traversal of $z_2$ on $FAS_2$ result in leaf node $v_2$. Then, we have $t_{v_1} = t_{v_2}$.

Thus, the algorithm selected by $FAS_1$ on $z_1$ is the same as the algorithm selected by $FAS_2$ on $z_2$.

We prove Theorem 6.4.1 after showing the following lemma. Recall that in Section 6.4.2 we defined $FAS_1$, and $FAS_2$ trained on infinite training sets, with different epsilon values. We also define a $\epsilon$-stable bijection. A bijection $f_{\epsilon, \epsilon'} : D \rightarrow D$ is a $\epsilon$-stable bijection if for $f_{\epsilon, \epsilon'}(s \cdot p) = s' \cdot p$, any workload $W$, and a scale/$\epsilon$-exchangeable algorithm $A$:

$$\text{error}(A, W, s p, \epsilon) = \text{error}(A, W, s' p, \epsilon')$$

**Lemma 6.4.2.** Let $f_{\epsilon, \epsilon'}$ an $\epsilon$-stable bijection. We denote the nodes of $FAS_1$ at level $i$ as $v_1^i, \ldots, v_2^i$, and similarly for $FAS_2$: $w_1^i, \ldots, w_2^i$. Then $\forall i, j$: $V_j^i = f[W_j^i]$ and $t_{v_j^i} = t_{w_j^i}$.

83
Algorithm 5 Pythia($W, x, \epsilon, \rho$)

1: $\epsilon_1 = \rho \cdot \epsilon$
2: $\epsilon_2 = (1 - \rho) \cdot \epsilon$
3: $d = \text{NNZ}(\Delta F)$
4: $f_z = F(W, x, \epsilon)$
5: $\tilde{f}_z = f_z + \Delta F^T \text{Lap}(d/\epsilon_1)$
6: $A^* = \text{FAS}(\tilde{f}_z)$
7: $\hat{y} = A^*(W, x, \epsilon_2)$
8: return $\hat{y}$

Proof. The infinite size of the training data as well as the scale/\epsilon exchangeability of the algorithms in the labels guarantee that both roots of FAS$_1$ and FAS$_2$ share the same label distribution. Consider the first split of FAS$_1$: $(v_1, v_2)$, we know that this split achieves the highest impurity improvement: $\theta_1$. We argue that the first split of FAS$_2$: $(W_1, W_2)$ is such that $V_1 = f[W_1], V_2 = f[W_2]$, if it was any other case then the impurity improvement would be less in either FAS$_1$, or FAS$_2$. Because of $f$ is an $\epsilon$-stable bijection this also implies that $t_{v_1} = t_{w_1}$ and $t_{v_2} = t_{w_2}$. As tree construction is made top-down, we recursively apply the same argument and the proof follows.

Proof. [Theorem 6.4.1] From Lemma 6.4.2 we have that all non-leaf nodes $v^i_j$ and $w^i_j$ make a split on the same feature, more specifically $\forall f \in \mathcal{F}\{scale\}$ : the split condition is the same, and that for $f = scale$ the split conditions are of the form $(f, s)$ and $(f, se_1/\epsilon_2)$ for FAS$_1$ and FAS$_2$ respectively.

This means that at traversal time, $z_1$ and $z_2$ will end up in the leaves $v^i_j$ and $w^i_j$ of FAS$_1$, and FAS$_2$. The proof follows from Lemma 6.4.2. \qed

6.5 Deploying Pythia

Pythia is a meta-algorithm with the same interface as a differentially private algorithm: its input is a triple $z = (W, x, \epsilon)$, and its output is $y$, the answers of $W$ on $x$ under $\epsilon$-differential privacy. Pythia works in three steps: feature extraction, algo-
algorith selection, and algorithm execution. First, using $\epsilon_1$ privacy budget it extracts a differentially private estimate of the features $\tilde{f}_z$ from the input $z$. Then based on $\tilde{f}_z$ it uses its FAS to choose an algorithm $A^*$, which runs with input $(W, x, \epsilon_2)$ and returns the result.

In Algorithm 5 we see an overview of Pythia. In lines 2-3 of Algorithm 5 we split the privacy budget to $\epsilon_1$ and $\epsilon_2$ to be used for feature extraction and algorithm execution, respectively. In line 4 we compute the number of total features that need to be privately computed ($\text{NNZ}$ is a function that returns the number of non-zero elements of a given vector). In line 5 we extract the true features $f_z$ and in line 6 we use the Laplace Mechanism to produce a private estimate $\tilde{f}_z$. In line 7 we apply the FAS on the noisy features $\tilde{f}_z$ and we get the chosen algorithm $A^*$. In line 8 we run $A^*$ with input $z = (W, x, \epsilon_2)$ and return the answer.

**Feature Extraction** Delphi provides Pythia with the set of features $F$ of the input $z = (W, x, \epsilon)$. As a reminder, features extracted from the sensitive dataset $x$ might potentially leak information about $x$; for that reason we need to privately evaluate the values of these features on $x$. To do so, we use the vector of sensitivities $\Delta F$ of each individual feature. We add noise to the features in the following manner: we assign a privacy budget $\epsilon_1$ for feature extraction, and then use the Laplace Mechanism to privately evaluate each feature’s value by using a fraction $\epsilon_1/d$ for each feature, where $d$ is the total number of sensitive features. This process guarantees that feature extraction satisfies $\epsilon_1$-differential privacy.

6.5.1 **Deployment Optimizations**

The first optimization we consider is *dynamic budget allocation*, and the second is *post-processing via noisy features*. In Algorithm 6 we show Pythia utilizing both optimizations. We now give an overview of each optimization.
Algorithm 6 PYTHIA($W, x, \epsilon, \rho$) – w/ Optimizations

1: $\epsilon_1 = \rho \cdot \epsilon$
2: $\epsilon_2 = (1 - \rho) \cdot \epsilon$
3: $d = \text{NNZ}(\Delta F)$
4: $f_z = F(W, x, \epsilon)$
5: $\tilde{f}_z = f_z + \Delta F^T \text{Lap}(d/\epsilon_1)$
6: $A^*, \tilde{f}'_z = \text{FAS}(\tilde{f}_z)$
7: $\epsilon'_2 = \epsilon_2 + (d - |\tilde{f}'_z|)/d \epsilon_1$
8: $\bar{y} = \text{OPTIMIZE}(\tilde{y}, W, \tilde{f}'_z)$
9: $\tilde{y} = \text{OPTIMIZE}(\tilde{y}, W, \tilde{f}'_z)$
10: return $\tilde{y}$

Dynamic Budget Allocation The first optimization we consider is to dynamically reallocate the privacy budget between feature extraction and the execution of the selected algorithm. Recall that the feature extraction step of Pythia consumes privacy budget $\epsilon_1$ to recover $d$ sensitive features from $x$. Then $\tilde{f}_z$ is used to traverse the decision tree FAS to choose an algorithm $A^*$. In reality, not all features are necessarily used at the tree traversal step. For example, in Fig. 6.3, while there are 2 sensitive features (scale, number of non-zero counts) in the FAS, any input traversing that FAS will only utilize one sensitive feature (either scale, or NNZ). In this example we have spent $\epsilon_1/2$ to extract an extra sensitive feature that we do not use.

Dynamic Budget Allocation recovers the privacy budget spent on extracting features that are not utilized in the tree traversal step and instead spends it on running the chosen algorithm $A^*$. More specifically, given $d' < d$ sensitive features were used to traverse the tree, we update the privacy budget of the algorithm execution step to $\epsilon'_2 = \epsilon_2 + (d - d')/d \cdot \epsilon_1$. Lines 7 and 8 of Algorithm 6 reflect this optimization. In the example of Fig. 6.3 this means that we will run the chosen algorithm with privacy budget $\epsilon_2 + \epsilon_1/2$ and thus achieve higher accuracy on the release step.

Post-Processing via Noisy Features The second deployment optimization we propose is a post-processing technique on the noisy output $\tilde{y}$ of Pythia by reusing the noisy
features. The intuition behind our method is the following, the true features extracted from the dataset $f_z$ impose a set of constraints on the true answers of the workload $y$. We describe these constraints as a set $C$, i.e., $y \in C$. Since $\tilde{y}$ is a noisy estimate of $y$, it might be the case that $\tilde{y} \notin C$. In the case that $C$ is a convex set, we can project the noisy answer to $C$ and get another estimate: $\bar{y} = Proj_C(\tilde{y})$, where $Proj_A(x) \triangleq \arg \min_{y \in A} \|x - y\|$. Doing this guarantees that the error of $\bar{y}$ will be smaller than $\tilde{y}$.

**Theorem 6.5.1.** Let a convex set $C$, and points $y, y'$ where $y \in C$. Then $\|y - y^*\|_2 \leq \|y - y'\|_2$ where $y^* = Proj_C(y')$.

At deployment time we do not know the true features $f_z$, instead we have a noisy estimate $\tilde{f}_z$. We overcome this challenge by creating a relaxed convex space $\tilde{C}$ based on the noisy features and project to that. As an example, consider dataset $x$ and workload $W = I$ the identity workload, at run-time suppose that the scale $\tilde{s}_z$ is used. Then we create the constraint $\|y\|_1 \leq \tilde{s}_z + \xi$, where $\xi \sim 1/\epsilon_1$ is a slack parameter, to account for the noise added. Lastly we project the noisy answer $\tilde{y}$ to space defined by our constraint. We show experimentally significant improvements in the quality of the final answer $\tilde{y}$ using this technique.
In this chapter we present our experimental evaluation of the systems presented in this thesis. We evaluate our systems using both real and benchmark datasets on a variety of different use cases. Our main focus is reporting the error incurred for a given privacy level. More specifically, we evaluate the end-to-end performance of the proposed systems in a variety of different settings – i.e., privacy levels, workload size, data size, etc. Additionally, we compare our proposed algorithms with the current state-of-the-art competitor algorithms, showing improvements over prior work. We also perform a system analysis of both PrivSQL and Pythia, by changing each factor one at a time and controlling the input configurations.

The chapter is divided in two distinct sections, in Section 7.1 we present the empirical evaluation of PrivSQL and in Section 7.2 the evaluation of Pythia. In Section 7.1.2 we present the end-to-end error evaluation of PrivSQL on a real world use case and a benchmark, for the former we show that for more than 60% of the queries evaluated PrivSQL offers less than 10% relative per query error. In Section 7.1.3, we compare PrivSQL with prior work (FLEX[JNS18]) where we show that PrivSQL offers an average case improvement in total error incurred of 2 orders
of magnitude – which can go up to 10 orders of magnitude for certain queries. Our comparison with extends by running PrivSQL in “single query mode”, where again we show improvements of at least 2 orders of magnitude across all queries. Lastly, in Section 7.1.4 we evaluate alternative choices for components of PrivSQL and offer an evaluation on the effect of truncation in the overall error incurred.

In Section 7.2.2 we evaluate Pythia for answering a workload of queries on 1- and 2-dimensional datasets. Our main finding is that across a multitude of inputs Pythia offers on average 60% improvement against the best “blind choice” algorithm – i.e., using the same algorithm across all inputs. In Section 7.2.3 we use Pythia as a building block for implementing a differentially private naive Bayes classifier (NBC), where we show Pythia offers competitive misclassification rates with that of a non-private baseline.

7.1 PrivSQL Evaluation

We evaluate PrivSQL on both a use case inspired by U.S. Census data releases as well as the TPC Benchmark H(TPC-H)[TPC93]. In Section 7.1.2 we present an end-to-end error evaluation analysis. In Section 7.1.3, we compare with prior work (FLEX[JNS18]). Lastly, in Section 7.1.4 we evaluate alternative choices for components of PrivSQL.

7.1.1 Setup

Table 7.1 summarizes settings with defaults in boldface.

Datasets: We use the public synthetic U.S. Census dataset [SASV17] with the following schema: PERSON(ID, SEX, GENDER, AGE, RACE, HID) and HOUSEHOLD(HID, LOCATION). We create two datasets from the full Census data by filtering on location: CENSUS\textsubscript{PM} limits to a specific PUMA region (a region roughly the size of a town) and CENSUS\textsubscript{NC} limits to locations within North Carolina. CENSUS\textsubscript{PM} con-
Table 7.1: PRIVSQL and input options used.

<table>
<thead>
<tr>
<th>Census Input</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>CENSUS\textsubscript{NC}, CENSUS\textsubscript{PM}</td>
</tr>
<tr>
<td>Privacy Policy</td>
<td>Person, Household</td>
</tr>
<tr>
<td>Privacy Budget $\epsilon$</td>
<td>2.0, 1.0, 0.5, 0.25, 0.125</td>
</tr>
<tr>
<td>Representative Workload</td>
<td>$W_1$, $W_2$, $W'_1$, $W'_2$</td>
</tr>
<tr>
<td>Query Workload</td>
<td>$W_1$, $W_2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TPC-H Input</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>TPC-H</td>
</tr>
<tr>
<td>Privacy Policy</td>
<td>Customer</td>
</tr>
<tr>
<td>Privacy Budget $\epsilon$</td>
<td>2.0, 1.0, 0.5, 0.25, 0.125</td>
</tr>
<tr>
<td>Representative Workload</td>
<td>$W_3$</td>
</tr>
<tr>
<td>Query Workload</td>
<td>$W_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRIVSQL Config.</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUDGET\textsc{ALLOC}</td>
<td>W\textsc{SIZE}, W\textsc{SENS}, NAIVE, VSSENS</td>
</tr>
<tr>
<td>PRIV\textsc{SYNGEN}</td>
<td>W-nnls, Identity, Part</td>
</tr>
</tbody>
</table>

tains 50$K$ and 38$K$ tuples in Person and Household respectively, while CENSUS\textsubscript{NC} contains 5.4$M$ and 2.7$M$ tuples, resp. We also use the TPC-H benchmark with a schema consisting of 8 relations. We scaled the data to 150$K$, 1.5$M$, and 6$M$ tuples in the Customer, Order, and Lineitem tables respectively.

**Policies:** We use two policies for the Census schema, ($\text{Person}, \epsilon$) and ($\text{Household}, \epsilon$) where the private object is a single individual, or a household, respectively. For the TPC-H schema we used ($\text{Customer}, \epsilon$) policy, which protects the presence of customers in the database.

**Workload:** Summary File 1 (SF-1)\cite{Cen10} is a set of tabulations released by the U.S. Census Bureau. We parsed their description and constructed two workloads of SQL queries: $W_1$ and $W_2$. $W_1$ contains 192 complex queries, most of which contain joins and self joins on the base tables Household and Person as well as correlated subqueries. An example query is the “Number of people living in owned houses of size 3 where the householder is a married Hispanic male.” The second workload $W_2 \supset W_1$ includes an additional 3,493 linear counting queries on Person relation. An example
Table 7.2: View Statistics for queries of $W_2$.

<table>
<thead>
<tr>
<th>View Group</th>
<th># of Queries</th>
<th>Person policy</th>
<th></th>
<th></th>
<th>Household policy</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># of</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sens</td>
<td>Median QERROR</td>
<td>Sens</td>
<td>Median QERROR</td>
<td></td>
</tr>
<tr>
<td>#1</td>
<td>23</td>
<td>0</td>
<td>0.0</td>
<td>1</td>
<td>948.1</td>
<td></td>
</tr>
<tr>
<td>#2</td>
<td>3575</td>
<td>1</td>
<td>85.4</td>
<td>4</td>
<td>400.6</td>
<td></td>
</tr>
<tr>
<td>#3</td>
<td>25</td>
<td>2</td>
<td>636.4</td>
<td>8</td>
<td>30,474.2</td>
<td></td>
</tr>
<tr>
<td>#4</td>
<td>8</td>
<td>4</td>
<td>5,916.6</td>
<td>16</td>
<td>8,484.8</td>
<td></td>
</tr>
<tr>
<td>#5</td>
<td>12</td>
<td>6</td>
<td>5,294.7</td>
<td>24</td>
<td>42,056.4</td>
<td></td>
</tr>
<tr>
<td>#6</td>
<td>6</td>
<td>17</td>
<td>17,362.2</td>
<td>68</td>
<td>34,670.4</td>
<td></td>
</tr>
<tr>
<td>#7</td>
<td>36</td>
<td>25</td>
<td>8,413.9</td>
<td>100</td>
<td>40,860.3</td>
<td></td>
</tr>
</tbody>
</table>

Linear query is the “Number of males between 18 and 21 years old.”. For evaluation of TPC-H we used queries $q_1, q_4, q_{13}, q_{16}$ from the benchmark to derive $W_3$ a workload of 61 queries, by expanding on the GROUP BY clause of the original queries.

**PrivSQL configuration:** The synopsis generation and budget allocation are configurable, as described in Section 5.4 and listed in Table 7.1. For the LearnThreshold algorithm described in Section 5.2.3, we set threshold as $\theta = 0.9$ and budget as $\epsilon_{mf} = 0.05 \cdot \epsilon$.

**Error Measurement:** For a query $q$, let $y = q(D)$ be its true answer, and $\tilde{y}$ be a noisy answer, we define the absolute error of $\tilde{y}$, as: $QERROR(y, \tilde{y}) = |y - \tilde{y}|$. Similarly, we define the relative error as: $RELERROR(y, \tilde{y}) = |y - \tilde{y}| / \max(50, y)$. In all experiments, we run each algorithm for 10 independent trials and report the average of the error function.

### 7.1.2 Overall Error Analysis

We evaluate PrivSQL on datasets $\text{Census}_{PM}$ and $\text{Census}_{NC}$ using workloads $W_1$ and $W_2$ and both Person and Household. Then we evaluate on TPC-H with the $W_3$ workload and Customer policy.
Figure 7.1: Relative error rates of PrivSQL. Top is $W_1$ on the CENSUS$_{NC}$ dataset for Person and Household policies. Bottom is $W_2$ on CENSUS$_{NC}$ for Person policy and $W_3$ on the TPC-H. Error rates stratified by true query answer size.

Error Rates: Figs. 7.1 and 7.2 summarize the RELERROR distribution of PrivSQL across different input configurations, stratified by the true query answer sizes. In each figure we draw a horizontal solid black line at $y = 1$, denoting relative error of 100%. A mechanism that always outputs 0 would achieve this error rate.

PrivSQL achieves low error on a majority of the queries. For the Person policy and CENSUS$_{NC}$ dataset (Figs. 7.1a and 7.1c), PrivSQL achieves at most 2%
RelError on 75% of the $W_1$ queries and at most 6% RelError on 50% of the $W_2$ queries. For the Household policy (Fig. 7.1b) all error rates are increased. The noise necessary to hide the presence of a household is much larger as removing one household from the dataset affects multiple rows in the Person table. PrivSQL also offers high accuracy answers for the $W_3$ workload on the TPC-H benchmark, where more than 60% of the queries achieve less than 10% relative error (Fig. 7.1d).

Fig. 7.2a shows error on the CensusPM dataset, using workload $W_1$ workload and Person policy. The trends are similar to the CensusNC case, but the error is higher as query answers are significantly smaller on CensusPM than on CensusNC. Fig. 7.2b shows more results on the CensusNC, across varying $\epsilon$ values. As expected, PrivSQL incurs smaller error higher values of $\epsilon$. We omit figures for other configurations due to space constraints.

Queries with smaller true answer sizes and higher sensitivity incur high error. We discuss these effects next.

Error vs Query Size: In Fig. 7.1 and Fig. 7.2a the results are grouped by the
size of the true query answer. The number of workload queries in each group is \{0 - 10^3 : 24, 10^3 - 10^4 : 73, >10^4 : 93\} for \(W_1\) and \{0 - 10^3 : 1869, 10^3 - 10^4 : 811, 10^4 - 10^5 : 742, >10^5 : 253\} for \(W_2\). Queries with size <10^3 have the highest error. As the true answer size increases, the error drops by an order of magnitude. Under the Person policy, 95\% of queries in \(W_1\) and \(W_2\) with size >10^3 have error <10\%. The median error for queries in \(W_1\) with true answer >10^4 is <.1\%. This further highlights the real-world utility of PrivSQL.

High error rates are mostly caused by queries with small true answer. Moreover, we observe a dramatic downwards error trend as the size increase for both \(W_1\) and \(W_2\). For instance, in the case of \(W_1\), 95\% of queries with size > 1,000 have error rate less than 10\% and 75\% of queries with size > 100k have error less than 0.1\%. These results further highlight the applicability of PrivSQL on an employment in a real world scenario.

**View Sensitivities:** In Table 7.2 we show statistics about the views generated from PrivSQL for workload \(W_2\), dataset Census\_NC, and both Person and Household policies. Rows of the table correspond to groups of views that have the same sensitivity. The second column shows the number of queries that are answerable from views in the group. The rest of the table summarizes the sensitivity of views in each group and the median absolute error (QError) across queries answerable from these views under Person and Household policy, resp. For instance, there are 3575 queries answerable by views with sensitivity 1 under Person policy, and have a median absolute error of 85.

We see that as the view sensitivity of a group increases so does the median QError across queries. The connection is not necessarily linear due to choices in PrivSynGen and BudgetAlloc. We also see that, for the same group, the Household policy leads to higher sensitivity bounds and higher error rates. This is
because the removal of a single row in the Household table affects multiple rows in Person.

We also derived the equivalent view statistics for TPC-H. For $W_3$ PRIVSQL creates 4 views with computed sensitivities: 0, 104, 182, 390 and QERROR values are: 0, 111, 112K, 3.5K respectively. Again we see that the sensitivity to error connection is non-linear due to factors like truncation.

7.1.3 Comparison with Prior Work

We next compare with FLEX [JNS18], though a direct comparison is difficult for several reasons. FLEX is designed for answering one query at a time, while PRIVSQL answers multiple queries under a common budget. FLEX satisfies $(\epsilon, \delta)$-differential privacy, a relaxation of DP, whereas for PRIVSQL, $\delta = 0$. PRIVSQL supports multiple privacy policies, while FLEX does not (and specifically cannot support the Household policy). We set $\delta = 1/n$ for FLEX, where $n$ is the number of rows in the Person table, and consider the Person policy.

For our first comparison, we compare PRIVSQL against BASELINE$\_FLEX$, a natural extension of FLEX adapted for answering a workload of queries, where the privacy budget is evenly divided across the set of answered queries. Then, we provide a more direct “apples to apples” comparison by (a) running both systems one query at a time and (b) comparing their sensitivity engines.

**Workload Query Answering** We evaluate performance on workloads $W_1$ and $W_2$ on CENSUS$_{NC}$ dataset. FLEX does not support 42 queries of $W_1$, which are complex queries containing correlated subqueries. We omit these from the evaluation. In Fig. 7.3 we present the results, with error distributions again stratified by query size. We draw a solid black line at RELERROR = 1, which corresponds to the error of the “just report zero” baseline [McS18]. For the $W_1$ workload, the BASELINE$\_FLEX$ relative error rate exceeds 1 for more than 75% of the queries, while PRIVSQL has error less
than 2% for 75% of the queries. Even for large query sizes ($>10^4$), BASELINE$_{Flex}$ has high error rates, as $W_1$ mostly contains complex queries with high sensitivity. In the case of small query size we can see that PrivSQL has significantly lower error than BASELINE$_{Flex}$ and offers an improvement over the all-zeros baseline on half the queries. For the $W_2$ workload (Fig. 7.3b) the trends are similar.

The above experiments compare the systems in terms of error on an entire workload. One factor that contributes to PrivSQL achieving comparably lower error than the baseline extension of FLEX is that it has more sophisticated support for workloads: VSELECTOR groups together queries which may compose parallely and enjoy a tighter privacy analysis, and techniques like W-nnls in the synopsis generator use least squares inference to further reduce the error of query answers.

**Single Query Answering** As discussed earlier, FLEX is designed as an on-line query answering system where each query is privately estimated and returned to the user separately under its own privacy guarantee $\epsilon_q$. To provide a more direct comparison with FLEX, we run our system in “single query mode”, denoted by PrivSQL$_{sqm}$,
which takes as input a workload containing a single query and returns a private syn-
opsis to answer that query. We evaluate both systems on workload $W_1$ on CensusNC and Person policy and use a per-query budget of $\epsilon_q = 0.01$. We omit showing results for queries in $W_2 \setminus W_1$ as those queries have the same sensitivity, and hence same error under both systems.

This evaluation allows us to decouple error improvements due to workload-related components – such as VSelector, BudgetAlloc, and PrivSynGen – and focus on the query analysis components SensCalc and VRewrite.

Fig. 7.4 shows for each query the QERROR of Flex on the y-axis and the QERROR of PrivSQL$_{sqm}$ on the x-axis. Queries are grouped together w.r.t. their computed sensitivity under SensCalc. Groups #6 and #7 are queries with correlated subqueries and are unsupported by Flex. However, for illustration purposes, we allow Flex to use the de-correlation techniques of VSelector in order to answer them. All queries lie over the dotted $x = y$ diagonal line, i.e., for every query, PrivSQL$_{sqm}$ offers lower error than Flex. This improvement is over 10 orders of magnitude for some Flex supported queries (Group 5). All improvements are due to two factors: (a) the tighter sensitivity bounds of SensCalc compared with Flex rules and (b) the VRewriter truncation technique which helps bound the global sensitivity, avoiding the need for smoothing.

Next, we isolate the sensitivity engines of both Flex and PrivSQL and compute only the sensitivity bounds (without truncation or smoothing). In Fig. 7.5 we show our results using the same groups as Fig. 7.3. For all queries SensCalc offers a strictly better sensitivity analysis with improvements ranging up to $37 \times$ on Flex supported queries. For group #2 that contains > 40% of the $W_1$ queries, SensCalc offers an improvement of $4 \times$. 

97
7.1.4 System Analysis

Next, we perform a series of experiments evaluating the performance of PrivSQL with different BudgetAlloc and PrivSynGen options. In Fig. 7.6 we show results of PrivSQL where we change component instantiations one-at-a-time in order to better understand their impact. We also analyze the effect of the truncation operation in terms of overall error (see Fig. 7.7). The next evaluations are on workload $W_1$ on Census$_{NC}$ and Person policy.

Effect of Budget Allocator: In Fig. 7.6a we show the absolute error distribution of PrivSQL for different BudgetAlloc choices. WSIZE and WSENS offer the best error rates, with comparable performance. This is due to low composition parallelism between queries of each partial workload $Q_V$. High error rate queries perform similarly across BudgetAlloc instantiations. This further explains that NAIVE outperforms VSENS, as the latter assigns a larger privacy budget to high sensitivity views, for which their sensitivity dominates the error factor.

Effect of Synopsis Generator: In Fig. 7.6b we show the absolute error distribution of PrivSQL for different PrivSynGen choices. For representative workload $W_1$
(left of the dotted line), we see that W-NNLS outperforms the other 2 methods. The non-negative least squares inference technique offers significant advantage since it optimizes for the exact queries that the analyst submits.

**Effect of Representative Workload:** We create $W'_1$, a smaller representative workload of 35 queries that capture the join structures of queries in $W_1$. The change in representative workload only affects the W-NNLS synopsis generator, as Identity and Part are workload agnostic (Section 5.4). The results show that the performance of W-NNLS deteriorates when $W'_1$ is used instead of $W_1$ (Fig. 7.6b, right of the dotted line). This suggests that data owners with little knowledge about analyst queries may prefer to instantiate PrivSQL with Identity or Part.

**Effect of Representative Workload:** We create $W'_1$ and $W'_2$, two smaller representative workloads that can be used to answer queries from $W_1$ and $W_2$ respectively. In Fig. 7.6b we show results for $W'_1$ alone, which contains 35 queries. Results for $W'_2$ are omitted due to space constraints. As discussed in Section 5.4, Identity and Part are workload agnostic, while W-NNLS is workload aware. For that reason,
we only show W-nnls for input $W'_1$ (right of the dotted line), since IDENTITY and PART have identical error rates for either $W_1$ or $W'_1$ representative workloads.

The performance of W-nnls($W'_1$) deteriorates and now both PART and IDENTITY offer better error rates that are comparable to that of W-nnnls($W_1$). This allows data owners with little to no knowledge about analyst queries to instantiate PrivSQL with IDENTITY or PART with little loss in analyst accuracy.

**Effect of Truncation Operator:** The truncation rewrite operation of VREWRITER might introduce bias in the synopses generated – due to tuples being dropped from the base tables. To quantify this bias, we isolate the queries for which Algorithm 2 adds a truncation operator in the query plan of their corresponding view. For all queries in our workloads, the truncated attribute is hid in Person and in PrivSQL the LEARNTHRESHOLD as described returns w.h.p. a threshold value of 4. For those queries and for different truncation levels, we measure their total error as well as their bias due to the addition of truncation in their corresponding views. In Fig. 7.7 we summarize our results. In both figures the x-axis is labeled by the truncation
Figure 7.8: Relative error and bias distributions of truncation-affected queries only, for different truncation values. Numbers in parentheses denote the percentage of tuples truncated at the corresponding value.

Small truncation values imply less noise (tighter view sensitivity bounds) but more dropped tuples. For small truncation values, bias dominates overall error. However, note that some queries have 0 bias even for truncation value 1 (e.g., counting households with a single person is not affected by a truncation value of 1). As the truncation value increases, the boxplots narrow but also rise. They narrow because the high error queries improve as their main source of error is bias which drops with increasing truncation value. They rise because increasing the truncation value causing more noise to added to query answers, hurting low error queries. Next, we observe a trade-off between high and low error queries, with high error queries being favored from high truncation values. More specifically, high error rates are dominated by the bias term – e.g., there is a change of 2 orders of magnitude between truncation values for the 95 percentile error queries. On the other hand, smaller error rates are mostly affected by the added noise. Empirically, we see that a truncation
choice between 4 and 6 offers the best of both worlds.

7.2 Pythia Evaluation

In our experimental evaluation we consider two different tasks: 1D and 2D range queries. For each task we train a single version of Pythia that is evaluated on all use cases for that task. We consider the standard use case of workload answering and we also demonstrate that Pythia can be very effective for the use case of building a multi-stage differentially private system, specifically a Naive Bayes classifier.

In Pythia we always set $\rho = 0.1$ to split the privacy budget for the feature extraction step. Tuning the budget allocation between the two phases is left for future work. For algorithms used by Pythia, we parameterized using default values whenever possible.

**Summary of Results** We evaluate performance on a total of 6,294 different inputs across multiple tasks and use cases. Our primary goal is to measure Pythia’s ability to perform algorithm selection, which we measure using regret. Our main findings are the following:

- On average, Pythia has low regret ranging between 1.27 and 2.27. If we compare Pythia to the strategy of picking a single algorithm and using it for all inputs, we find that Pythia always has lower average regret. This is indirect evidence that Pythia is not only selecting a good algorithm, on average, it is selecting different algorithms on different inputs.

- For the multi-stage use case, we learn a differentially private Naive Bayes classifier similar to Cormode [Cor11] but swap out a subroutine with Pythia. We find that this significantly reduces error (up to $\approx 60\%$). In addition, results indicate that for this use case Pythia has very little regret: it performs nearly as well as the (non-private) baseline of Informed Decision.
We also examine some aspects of the training procedure for building Pythia.

- We show that our regret-based learning technique using the group impurity measure results in lower average regret compared to the standard classification approach that uses the Gini impurity measure. The reduction is more than 30% in some cases.

- The learned trees are fairly interpretable: for example, the tree learned for the task of 2D range queries reveals that Pythia: selects DAWA when features suggest the data distribution is uniform or locally uniform, selects Laplace for small domains, and AHP for large scales.

In terms of run time, Pythia adds negligible overhead to algorithm execution: some algorithms take up to minutes for certain inputs, but Pythia runs in milliseconds. Training is somewhat costly due to the generation of training data (which takes about 5 hours). However, once the training data is generated, the training itself takes only seconds.

In Section 7.2.1, we describe the inputs supplied to the training procedure Delphi. For each use case, we describe the setup and results in Sections 7.2.2 and 7.2.3. Section 7.2.4 illustrates the interpretability of the Feature-based Algorithm Selector and the accuracy improvements due to our regret based learning procedure.

7.2.1 Delphi setup

Recall that Pythia is constructed by the Delphi training procedure described in Sections 6.4 and 6.4.2. To instantiate Delphi for a given task, we must specify the set of algorithms $A_T$, the inputs $Z_T$, and the features used.

**Algorithms** The set of algorithms $A_T$ is equal to the set of algorithms shown in Section 6.1.3, except for AGRID and DPCUBE, which were specifically designed for
data with 2 or more dimensions and are therefore not considered for the task of answering range counting queries in 1D.

**Inputs** We construct $\mathcal{Z}_T$, the set of triples $(W, x, \epsilon)$, as follows. The value of $\epsilon$ is fixed to 1.0, leveraging the optimization discussed in Section 6.4.2. The datasets $x$ are constructed using the methods described in Section 6.4.2, with the parameters set as follows: $\mathcal{D}_{public}$ consists of datasets for a given task as described in Table 7.3; the set of scales is set to $S = \{2^5, 2^6, \ldots, 2^{24}\}$; and the set of domain sizes is $K = \{128, 256, \ldots, 8192\}$ for 1D and $K = \{4 \times 4, 8 \times 8, \ldots, 128 \times 128\}$ for 2D. This yields 980 datasets for the 1D task and 1080 datasets for 2D.

The workload $W$ comes from the set of representative workloads, $W_T$, which varies by task. For 1D, we use 2 representative workloads: *Identity* is the set of all unit-length range queries; and *Prefix* is the set of all range queries whose left boundary is fixed at 1. For 2D, we use 4 workloads, each of consisting of 1000 random range queries, but differing in permitted lengths. The *Short* workload has queries such that their length $m$ satisfies $m < d/16$ for domain size $d$, *Medium* has $d/16 \leq m < d/4$, *Long* has $m \geq d/4$ and *Mixed* consists of a random mix of the previous types.

By taking every combination of workload, dataset, and $\epsilon$ described above, we have $2 \times 980 \times 1 = 1,960$ inputs for 1D and $4 \times 1080 \times 1 = 4,320$ inputs for 2D. For each input, we run every algorithm in $\mathcal{A}_T$ on it 20 times (with different random seeds) and estimate the algorithm’s error by taking the average across random trials. We use this to empirically determine the regret for each algorithm on each input.

**Features** Recall that in Delphi, each input $(W, x, \epsilon)$ is converted into a set of features. The *dataset features* and their corresponding sensitivities are as follows:

- The *domain size*, denoted $d$. This feature has sensitivity zero because the domain size of neighboring datasets is always the same, i.e., the domain size of
Table 7.3: Overview of the datasets used for each task $T$.

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Domain Size</th>
<th>Original Scale</th>
<th>Prior Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADULTFRANK</td>
<td>4,096</td>
<td>32,561</td>
<td>[HLM12],[LHMW14]</td>
</tr>
<tr>
<td>HEPHT</td>
<td>4,096</td>
<td>347,414</td>
<td>[LHMW14]</td>
</tr>
<tr>
<td>INCOME</td>
<td>4,096</td>
<td>20,787,122</td>
<td>[LHMW14]</td>
</tr>
<tr>
<td>MEDCOST</td>
<td>4,096</td>
<td>9,415</td>
<td>[LHMW14]</td>
</tr>
<tr>
<td>NETTRACE</td>
<td>4,096</td>
<td>25,714</td>
<td>[ACC12],[HRMS10],[ZCX+13],[ZCX+14b]</td>
</tr>
<tr>
<td>SEARCHLOGS</td>
<td>4,096</td>
<td>335,889</td>
<td>[ACC12],[HRMS10],[ZCX+13],[ZCX+14b]</td>
</tr>
<tr>
<td>PATENT</td>
<td>4,096</td>
<td>27,948,226</td>
<td>[LHMW14]</td>
</tr>
</tbody>
</table>

Task: 2D Range Queries

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Domain Size</th>
<th>Original Scale</th>
<th>Prior Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADULT-2D</td>
<td>256 x 256</td>
<td>32,561</td>
<td>[HLM12],[LHMW14]</td>
</tr>
<tr>
<td>BJ-TAXI-S</td>
<td>256 x 256</td>
<td>4,268,780</td>
<td>[HCA+15]</td>
</tr>
<tr>
<td>BJ-TAXI-E</td>
<td>256 x 256</td>
<td>4,268,780</td>
<td>[HCA+15]</td>
</tr>
<tr>
<td>SF-TAXI-S</td>
<td>256 x 256</td>
<td>464,040</td>
<td>[PSDG09]</td>
</tr>
<tr>
<td>SF-TAXI-E</td>
<td>256 x 256</td>
<td>464,041</td>
<td>[PSDG09]</td>
</tr>
<tr>
<td>CHECKING-2D</td>
<td>256 x 256</td>
<td>6,442,863</td>
<td>[HMM+16]</td>
</tr>
<tr>
<td>MD-SALARY-2D</td>
<td>256 x 256</td>
<td>70,526</td>
<td>[HMM+16]</td>
</tr>
<tr>
<td>LOAN-2D</td>
<td>256 x 256</td>
<td>550,559</td>
<td>[HMM+16]</td>
</tr>
<tr>
<td>STROKE-2D</td>
<td>256 x 256</td>
<td>19,435</td>
<td>[HMM+16]</td>
</tr>
</tbody>
</table>

A dataset is public information.

- The scale is defined as $S(x) = \|x\|_1$, and corresponds to the total number of tuples in the dataset. Since the absence or presence of any tuple in the dataset the scale can change at most by 1, we have $\Delta S = 1$.

- The number of non-zeros is $\text{NNZ}(x) = |\{x_i \in x | x_i \neq 0\}|$. Changing any tuple in $x$ alters the number of non-zeros by at most 1 so $\Delta \text{NNZ} = 1$.

- The total variation between the uniform distribution and $x$ is:

$$\text{tvd}_u(x) = \frac{1}{2} \sum_{i=1}^{d} |x_i - u|$$

where $u = \|x\|_1/|x|$. We have $\Delta \text{tvd}_u = 1 - \frac{1}{d} \leq 1$. 

105
• The partitionality of $x$ is denoted $\text{Part}$ and is a function that returns minimum cost partition of $x$ according to the partition score defined in Li et al. [LHMW14]. Given the analysis of Li et al. [LHMW14], it is straightforward to show that $\Delta \text{Part} = 2$. $\text{Part}$ has low values for datasets whose histograms can be summarized using a small number of counts with low error.

The workload features vary by task. For the task of 1D range queries, we use the binary feature “is the average query length less than $d/2$?” For 2D range queries, we use a feature that maps a workload to one of 4 types: short, medium, long, or mixed. If all queries are short then it is mapped to short, similarly for medium and long; otherwise, it is mapped to mixed. As discussed in Section 6.4.2, the workload feature is used at the root of the tree to map a test instance to the appropriate subtree. For 2D, workloads are mapped directly by the above function; for 1D, workloads with average query length of less than $d/2$ are mapped to the Identity subtree and the rest are mapped to the Prefix subtree. Workload features have sensitivity zero because they do not depend on the private input $x$.

### 7.2.2 Use Case: Workload Answering

We first consider answering a single workload of queries $W$ on a dataset $x$ given a fixed privacy budget of $\epsilon$. Our goal is to evaluate Pythia’s ability to select the appropriate algorithm for a given input. We measure this ability by calculating regret: given a test input $z = (W, x, \epsilon)$ we run each algorithm in the set $\{\text{Pythia}\} \cup A_F$ on this input 20 times using different random seeds and calculate average error for each algorithm. Average error is then used to derive regret with respect to $A_F$. Note that when Pythia is invoked without optimizations (see Algorithm 5), even if one assumes it chooses the best algorithm $A^*$ for an input $z$, its regret will be $> 1$. This is because Pythia has to execute $A^*$ for privacy budget $\epsilon_2 > \epsilon$. 

106
Datasets The test inputs that we use are drawn from the set $Z_T$, which was described in the previous section on training. Of course this poses an additional challenge: we should not evaluate Pythia on an input $z$ that was used in training. To ensure fair evaluation, we employ a kind of stratified $\ell$-fold cross-validation: $Z_T$ is partitioned into $\ell$ folds such that each fold contains all of the inputs associated with a common source dataset from $D_{\text{public}}$. This ensures that the training procedure does not have access to any information about the private datasets that are used in testing. The number of source datasets varies by task: as indicated in Table 7.3, for the 1D task, $|D_{\text{public}}| = 7$ and thus $\ell = 7$; for 2D, $|D_{\text{public}}| = \ell = 9$. Reported results are an aggregation across all folds.

Algorithms Compared We compare Pythia against the baselines presented in Section 6.2.1. More specifically, we compare against Informed Decision, which always achieves a regret of 1 but is non-private and Blind Choice, which uses a single algorithm for all inputs.

In addition, the optimizations described in Section 6.5.1 are used: budget reallocation is used for both 1D and 2D and post-processing is used for 1D only.

Results Fig. 7.9 shows the results for both tasks. Each bar in the “All” group corresponds to the average regret over all test inputs. The other bar groups report average regret over subsets of the test inputs based on workload type. The dotted line corresponds to Informed Decision with regret = 1. Algorithms whose average regret exceeds 10 were omitted, namely AHP, MWEM, PRIVELET, and UNIFORM for 1D and DAWA, MWEM, UNIFORM, and DPCUBE for 2D. Additionally, in Section 7.2.5 we provide more detailed results where we analyze the regret of different algorithms for fixed values of shape, domain size, and scale.

The results show that Pythia has lower average regret than all other techniques.
In addition, Pythia’s regret is generally low, ranging between 1.27 (Prefix 1D) and 2.27. (Short 2D). It is also interesting to see that among the single algorithm strategies, the algorithm with lowest regret changes depending on the subset of inputs: for example, Hb has lower regret than DAWA for 1D Identity workload whereas the opposite is true for the 1D Prefix workload. The results provide indirect evidence that Pythia is selecting different algorithms depending on the input and achieving lower error than any fixed algorithm strategy.

7.2.3 Use Case: Multi-Stage Task

In this section, we evaluate Pythia by building a multi-stage differentially private system, namely a Naïve Bayes Classifier (NBC) [MN98]. Fitting an NBC for binary classification requires computing multiple 1D histograms of possibly heterogeneous domain sizes and shapes. We use Pythia to automatically select the most appropriate algorithm to use for each histogram. We evaluate performance using two datasets from the UCI repository [Lic13] that, for the purposes of evaluating Pythia, represent two extreme cases: one has a small number of homogeneous histograms, the other has a larger number of more diverse histograms. This way we can see whether the
benefit of algorithm selection increases with the heterogeneity of the input.

Given a $k$-dimensional dataset, with attributes $\{X_1, \ldots, X_k\}$ and a binary label $Y$, an NBC requires computing a histogram on $Y$ and, for each attribute $X_i$, a histogram on $X_i$ conditioned on the value of $Y$ for each possible value of $Y$. In total, this requires estimating $2k + 1$ histograms. In addition, once the histograms are computed, they are used to fit a statistical model. We consider two different models: the Gaussian [Zha04] and Multinomial [MN98] models. To compute an NBC under $\epsilon$-differential privacy, each histogram can be computed using any differentially private algorithm provided it receives only an $\epsilon' = \epsilon/(2k + 1)$ share of the privacy budget.

Datasets The first dataset is the Skin Segmentation [BD12] dataset. Tuples in the dataset correspond to random pixel samples from face images of individuals of various race and age groups. In total there are $245K$ tuples in the dataset. Each tuple is associated with 3 features $R, G, B$ and the labels are $\{\text{Skin}, \text{NoSkin}\}$. The second dataset we use is the Credit Default dataset [YhL09] with $30K$ tuples. Tuples correspond to individuals and each tuple consists of 23 features consisting of demographic information of the individual, as well as her past credit payments and credit status. The binary label indicates whether or not the borrower defaults. Note that as a pre-processing step, we removed 7 features that were not predictive for the classification task. To get test datasets of diverse scales, we generate smaller datasets by subsampling. For Skin Segmentation, we sample three datasets of sizes $1K$, $10K$, and $100K$, and for Credit Default, two datasets of sizes $1K$ and $10K$.

Note that these datasets are used for testing only. Pythia is trained on different inputs, as described in Section 7.2.1.


**Algorithms compared** We are interested in evaluating how the choice of algorithm for computing each histogram affects the accuracy of the resulting classifier. We consider 5 ways of computing histograms: (1) non-private unperturbed histograms, (2) non-private Informed Decision, which for each histogram selects the algorithm that achieves lowest error, (3) Pythia, (4) the Laplace mechanism, and (5) DAWA. We evaluated these approaches for both Gaussian and the Multinomial NBCs. Note that NBC with the Laplace mechanism and Multinomial model corresponds to the algorithm proposed by Cormode [Cor11]. Accuracy is measured on a 50/50 random training/testing split. We repeat the process 10 times for different random trials and report the average misclassification rate across trials.

**Results** Figs. 7.10 and 7.11 report classifier error for the Gaussian and Multinomial NBCs respectively. The results indicate that Pythia achieves lower error than any other differentially private strategy. In many cases, it achieves error that is almost as low as that of Informed Decision, which is not private. Fig. 7.11 also indicates that an NBC built with Pythia outperforms the existing state of the art approach (Multinomial with Laplace) of Cormode [Cor11]. Somewhat surprisingly, Pythia is very effective even on the Skin Segmentation dataset whose histograms are fewer and homogeneous in terms of domain size. This is because Pythia almost always chooses Laplace for releasing the histogram on the label attribute (which has a domain size of 2) and DAWA for the the conditional distributions. This is close to the optimal choice of algorithms. Using Laplace or DAWA alone for all the histograms results in much higher error.

7.2.4 Evaluation of Training

We also examine some aspects of the training procedure for building Pythia.
Learned Tree  Fig. 7.13 illustrates the tree learned by Delphi for the task of 2D range queries on the Short workload. Internal nodes indicate a measured feature and leaves are labeled with the name of the algorithm that is selected for inputs that reach that leaf. The fraction shown in a leaf indicates for what fraction of those training inputs that were mapped to that leaf the selected algorithm was optimal. The tree can be fairly easily interpreted and offers insight into how Pythia chooses among algorithms. For instance, Pythia tends to select DAWA when measures indicate the
data distribution is uniform (low TVD) or locally uniform (low Partitionality). It tends to select Laplace for small domains, and AHP for large scales.

**Effect of Regret-based Learning** We also compare our approach of regret-based learning (Section 6.4.1), which uses Group Regret as its split criteria, against some alternatives including the standard Gini criterion measure, the Minimum Average Regret (MAR) and Regret Variance (VAR) criteria, all described in Section 6.4.1.

Fig. 7.12 compares these measures for the task of workload answering. The figure shows average error across the test inputs, exactly as was described in Section 7.2.2. It shows that the group impurity measure results in a roughly 30% reduction in average regret for 1D to the standard classification approach that uses the Gini
impurity measure. For 2D, the effect is less pronounced (14%) but still the group regret criterion achieves the lowest average regret.

7.2.5 Sensitivity Analysis

![Graph](image.png)

(a) Identity workload  
(b) Prefix workload

**Figure 7.14: Average Regret vs Shape**

Here we present additional experimental results that complement our analysis in Section 7.2.2. We further analyze the error incurred by algorithms for the task of workload answering for fixed values of shape, domain size, and scale.

In Fig. 7.14 we plot the average regret of each algorithm across different datasets, for the 1D tasks. Fig. 7.14a and 7.14b correspond to the identity and the prefix workload respectively. For the identity workload, Pythia has the lowest average regret amongst 5 data-sets and both AHP and DAWA have the lowest in 1 dataset. For the prefix workload, Pythia has the lowest average regret in 5 datasets and HB has the lowest regret in 2 datasets. The key point in this case is that when Pythia is not the best it is the second-best, which means that across datasets it has consistently good error.

In Figures 7.15 and 7.16 we see the corresponding plots when we fix the domain size and scale respectively, and then average out the regret measure. Again we see
similar trends, with Pythia being a consistently good choice.
In this chapter we present an overview of prior related work. We identify 4 broad categories that partition the space of prior work and organize the chapter accordingly.

8.1 Privacy Definitions

In [KM14] the authors propose Pufferfish privach, a semantic privacy framework which can among others can fully express differential privacy. The Pufferfish privacy framework allows experts to define novel privacy definitions by exposing underlying assumptions like adversarial background knowledge and the choice of privacy object. Much like PrivSQL this exposure allows data owners a greater flexibility in defining clearly the privacy semantics required by each application. However, and unlike PrivSQL tuning the privacy semantics is a non-trivial task since the authors use sets of probability distributions to express adversarial assumptions and privacy objects. Pufferfish can also be used to describe prior privacy definitions providing a better understanding on their assumptions. Despite its expressiveness and generality, Pufferfish has seen little adoption as it requires high expertise to correctly define the privacy semantics.
Blowfish privacy [HMD14] is a privacy framework inspired from Pufferfish privacy which lowers the barrier for authoring custom tailored privacy definitions. Blowfish privacy, much like Pufferfish privacy, allows the data owner to specify the information to be kept secret, i.e., the privacy object. However, and unlike Pufferfish, describing adversarial knowledge and privacy objects is significantly easier as both are described via a set of constraints. Then given a set of constraints, the privacy requirement can be expressed as a discriminative graph where nodes correspond to data values and edges connect nodes only if the respective data values are to be kept indistinguishable. For instance, the fully connected graph corresponds to differential privacy. Moreover, in [HMD15] the authors propose a general mechanism for authoring Blowfish algorithms given any discriminative graph.

In [HMA+17] the authors propose an instantiation of Pufferfish privacy tailored for a U.S. Census use case releasing aggregate employment statistics. More specifically, the authors consider the use case of linked employer-employee data and propose a custom privacy definition such that it satisfies the privacy requirements codified in U.S. legislature. Similar to the treatment in PrivSQL, the privacy definition allows for protection of employers or employees, giving flexibility on the protection provided from each query release. However, the algorithms proposed in that work are specific for counting queries over a single view of the data.

Another line of work closely connected with our privacy definition of Section 3.2 is edge-differential privacy [KRSY11] and node-differential privacy [KNRS13, DLL16, CZ13]. For a simple 2 relational schema with a single foreign key constraint both edge- and node-differential privacy could fully express the privacy semantics presented in this thesis. However, it is unclear how these definitions can apply to complex schemas with multiple integrity constraints resulting in more than 2 private base relations. We believe that the privacy semantics of PrivSQL are a strict generalization of edge- and node-differential privacy.
8.2 Single Query Answering

In the seminal work of [McS09a] the author proposes PINQ, a platform for data analysis under $\epsilon$-differential privacy. PINQ provides data analysts with a declarative language for submitting their queries on a sensitive database. The system then automatically analyzes and answers analyst queries such that the answers satisfy the specified privacy level. Much like PRIVSQL query answers are released outside a logical privacy firewall between the data analyst and the sensitive database. This completely removes the analyst from the privacy pipeline, allowing for protection against adversarial analysts (and not just honest-but-curious). In contrast with PRIVSQL, PINQ is a “one query at a time” system, meaning that once the total privacy budget is depleted, it stops answering incoming queries. Moreover, PINQ offers no support for optimizing the error across queries of the same view, like PRIVSQL offers. Lastly, PINQ does not support for privacy at multiple resolutions; it simply lets data owners to specify the maximum allowed privacy loss $\epsilon$.

In [JNS18] the authors propose FLEX an algorithm that can analyze and answer a single aggregate SQL query under $(\epsilon, \delta)$-differential privacy. In that work the authors introduce elastic sensitivity, an upper bound on the local sensitivity [NRS07] of a query and propose an efficient algorithm for computing the elastic sensitivity of a SQL query. In order to satisfy the privacy semantics and with the use of smoothing FLEX adds the appropriate noise the the true query answer. The query answering model implies that either the privacy loss is compounded over time or that the system needs to stop answering queries after a certain point. Moreover, FLEX does not support correlated subqueries in the SQL expressions. Finally, the privacy semantics of FLEX do not translate to real-world policies, or give any flexibility to the data owner.

In [AFG16] the authors study the problem of sensitivity estimation for counting queries on relational databases. First, they highlight that estimating the sensitivity
of a general relational algebra counting query is an undecidable problem. Their main finding is that for sensitivity estimation for conjunctive counting queries is computable, but becomes unbounded in the presence of join terms. The authors then propose bounds on sensitivity of conjunctive counting queries with databases with functional and cardinality dependencies.

In [CZ13] the authors propose the Recursive Mechanism, an algorithm for answering monotone SQL-like counting queries of high sensitivity. The main idea behind the algorithm is that it trades-off bias for variance. It does so by finding the a threshold that reduces the sensitivity of the query and then constructs a recursive sequence of lower sensitivity queries which can be used to approximate the input query.

An alternative notion of sensitivity called restricted sensitivity was introduced in [BBDS13]. Restricted sensitivity is used as an alternative to global sensitivity that can significantly lower the noise added. Like similar work, restricted sensitivity offers a bias/variance trade-off knob in the form of prior knowledge from the side of the analyst submitting queries. Instead of enumerating all neighboring databases to compute the global sensitivity, restricted sensitivity only enumerates over a database subspace, which satisfies the prior of the analyst. The authors provide with the machinery that given a query and a belief, they output another query with much smaller global sensitivity and return the noisy answer to that query instead. In the case that the sensitive data fits the analyst’s hypothesis, then the transformed query has the same answer with the original query. On the other hand, if the input data does not fit the hypothesis, then the transformed query has a different answer.

8.3 Multi-Query Answering

In a recent survey [HMM+16], Hay et al. compared 16 different algorithms for the task of answering a set of 1- or 2-dimensional range queries on a single table. The main finding of this work is that there is no single algorithm that dominates in terms
of errors for all tasks and data inputs. Even more importantly, 11 of the 16 algorithms in the study are data-dependent, meaning that the added noise (and therefore the resulting error rates) vary between different input datasets. Additionally, the authors show that for certain inputs, even traditionally “good” algorithms like DAWA [LHMW14] can be outperformed by simple baselines like the Laplace mechanism.

In [ZCP+14] the authors propose PrivBayes, a differentially private algorithm that given a data input and a privacy budget constructs a synthetic dataset. Construction of the synthetic data is done under differential privacy guarantees. Analysts can then use the synthetic dataset to submit an unbounded amount of queries all enjoying the same fixed privacy loss on the original data source. PrivBayes works by learning a Bayesian graphical model on the attributes of the original data source. It materializes under differential privacy the low dimensional marginal distributions defined from the graphical model. Lastly, using those marginals PrivBayes estimates the joint distribution of the data from which it draws tuples to generate the synthetic data. PrivBayes is a perfectly reasonable algorithm for answering batch of queries on a single relation. However, this technique does not extend to multi-relational schemas, since synthetic key generation would be prohibitively noisy under this model as shown also in [MPRV]

In [LMH+15] the authors propose Matrix Mechanism a more sophisticated approach for the problem of answering a set of linear counting queries on a single table. The Matrix Mechanism is one of the many algorithms that follow the select-measure-reconstruct paradigm. Under this paradigm and for a given workload of queries to be answered an algorithm first selects a new set of queries; then it computes differentially private answers to that set and finally answers the original input queries from performing inference on the noisy measurements. In Matrix Mechanism the query selection is done via solving optimization problem – i.e., minimizing the overall error in the original workload. Also note that Matrix Mechanism uses the vector represen-
tation for both queries and data tables, which can be a bottleneck in the presence of very high dimensional tables.

In a continuation of the Matrix Mechanism, the authors of [MMHM18] propose HDMM (High Dimensional Matrix Mechanism), an algorithm tailored for answering linear counting queries on high dimensional tables. HDMM much like the Matrix Mechanism also follows the select-measure-reconstruct paradigm. The main contribution of this work – and deviation from the Matrix Mechanism – is the implicit matrix representation that is used for the workload representation. This compact representation allows for an efficient search in the space of strategy workloads for finding one that can answer the original queries with high accuracy.

8.4 Other

Query answering using views is a well studied problem and we refer the reader to [Hal01] for an almost exhaustive survey of the space. The authors survey different approaches and applications to the problem of using views to answer queries over a database – by rewriting the queries in terms of the views.

In [HRMS10] the authors explore the problem of increasing the accuracy of noisy measurements over sensitive data by enforcing known constraints on the measurements. This problem is an instantiation of the problem of *inference on noisy data*. The authors provide an algorithm for releasing noisy prefix sum counts over an ordered domain. They show that their techniques provides meaningful improvements on the incurred error.

In [LT18] the authors propose new differentially private algorithms for the problem of private selection, i.e., selecting a candidate from a population based on sensitive data. More specifically, the authors present algorithms that have access to a scoring function over private candidates and privately select one instance that achieves high score. These algorithms are inspired from the sparse vector technique.
algorithm and are similarly parameterized by a threshold, which they try to optimize for. Algorithms presented in this work could be adapted to work in the context of algorithm selection in differential privacy. One way to do so, would be to set the regret of each algorithm as its scoring function. However, that would be extremely difficult as it requires calculating the sensitivity of the regret function across algorithms, as well as knowing a-priori all scoring functions as regret is dependent on the candidate population. Another approach would be to have another data-independent scoring function that only depends on the private outputs of the algorithms. Even with this approach the problem of sensitivity estimation remains. Lastly, the performance of all algorithms presented in this work are sensitive to the threshold choice which itself requires knowledge about the overall score distribution among candidates.
9.1 Thesis Summary

In this thesis we address the problem of data releases over traditional relational databases under rigorous privacy guarantees. We do so by proposing PrivSQL, which offers: (a) custom-tailored privacy semantics over relational data, (b) a unique and modular architecture, (c) view-based private synopsis generation for answering a rich class of SQL queries under fixed privacy loss, (d) state-of-the-art sensitivity estimation using truncation and rewriting techniques. We also examine the problem of releasing a single private synopsis, a task for which many different algorithms offer competitive error rates depending on data characteristics. We define the problem of algorithm selection and propose Pythia, a meta-algorithm that given a library of algorithms can choose the one that will incur the least error on a given input.

The first hurdle we need to overcome is to provide data owners with an intuitive, uncomplicated, but yet rigorous privacy definition. We do so by proposing differential privacy for multiple relations (see Chapter 3). Our definition is a novel generalization of differential privacy for relational databases with constraints. The main advantage of the proposed definition is that it allows data owners to easily
specify the privacy semantics necessary for their application by specifying a *privacy policy* in well understood terms of relational databases.

Our second contribution is the general architecture of PRIVSQL. The architecture proposed is driven from a set of principles and justifications informed by the rich privacy literature. Moreover, the modular architecture of PRIVSQL allows for future extensions and improvements as new research innovations are proposed. The architecture overview can be found in Chapter 4.

Our system supports answering multiple queries drawn from a rich class of SQL under a fixed privacy budget. To achieve that, PRIVSQL identifies a set of views over the schema of the relational database and generates private *synopses* for those views. Then queries submitted on the database are instead rewritten as queries over a view and answered from the corresponding synopsis. The central module that supports the view identification is VSELECTOR presented in Section 5.1.

PRIVSQL utilizes policy-aware view rewriting, truncation and constraint-obludious sensitivity analysis. These novel techniques guarantee that the private synopses generated from the views will provably ensure privacy as per the privacy policy and have high accuracy.

Lastly, we examine the problem of algorithm selection for releasing a single private synopsis. We present Pythia, a meta-algorithm that uses decision trees to privately select a suitable algorithm for a target input. The decision tree is learned via a novel regret-based learning method that is suitable for the algorithm selection problem.

9.2 Future Directions

One limitation of the current instantiation of PRIVSQL is that it does not account for updates in the input data, input queries, or the privacy policy. Currently, PRIVSQL releases a set of synopses tuned on a specific input 3-tuple: (data, queries, privacy policy). However, in real world deployments, database instances are ever-changing
due to additional data collection. Moreover, analyst interests change over time, resulting in different queries they submit on the sensitive data. Lastly, data owners might alter their privacy specifications over time – e.g., allowing for weaker privacy protection on older data.

These examples paint a more dynamic context, where the trinity of (data, queries, privacy policy) changes over time. We note that no $\epsilon$-DP algorithm can distinguish between answers to a single count query that differ by $\frac{1}{\epsilon} \log(1/\delta)$ with probability $1 - \delta$. That is, for $\epsilon = 0.1$, one can’t tell apart counts $x$ and $x + 13$ with 95% probability. This range increases as the number of queries increases. Thus, updating the private synopsis for every update to the database is unnecessary and a waste of privacy budget. This opens up a promising future direction where the problem to solve is how to update already-released private synopses under changes in either (a) the data, (b) the representative queries, or (c) the privacy policy.

A second limitation we identify is the subset of SQL queries supported. Currently, PrivSQL does not offer support for queries like SUM over a numerical attribute. Challenges for expanding the supported query language include both the sensitivity estimation as well as selecting views which can be re-used. For example, the SUM query can have very high sensitivity – even unbounded in the absence of some publicly known threshold. Moreover, adding support for a bigger class of SQL is non-trivial as it requires additional rules in the sensitivity calculator module such that sensitivity estimation is still tractable while still providing good upper bounds. Support for aggregate queries over numerical attributes could be achieved by using Lipschitz extension techniques. For example, the introduction of additional value-truncation operators for bounding the sensitivity of these queries. Additionally, the view selection module could be extended to add support for negation and existence operators by rewrites, which would require additional domain knowledge.

Lastly, the very architecture of PrivSQL opens an interesting future direction.
Taking a lesson from query optimizers for traditional relational database systems, an interesting future direction is the design of a privacy-aware query optimizer. This work, can be thought as an extension of our VREWRITER module, that given a query plan, sensitivity calculator, and a data source, it tries to find an alternative rewritten plan such that answering the rewritten plan on the data source incurs less error. This can be a strict improvement, e.g., due to better sensitivity estimation from the rule-based sensitivity calculator. The improvement can also be data-dependent, e.g., due to addition of truncation operators. Our proposed VREWRITER only scratches the surface of this complex problem. We believe that a more rigorous approach to plan rewriting can significantly increase the performance of a PRIVSQL-like system.
Bibliography


[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the*


