Towards Practical Privacy-Preserving Data Analytics

by

Noah Michael Johnson

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor David Wagner
Professor Deirdre Mulligan

Fall 2018

Towards Practical Privacy-Preserving Data Analytics

Abstract

Towards Practical Privacy-Preserving Data Analytics

by

Noah Michael Johnson

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Dawn Song, Chair

Organizations are increasingly collecting sensitive information about individuals. Extracting value from this data requires providing analysts with flexible access, typically in the form of databases that support SQL queries. Unfortunately, allowing access to data has been a major cause of privacy breaches.

Traditional approaches for data security cannot protect privacy of individuals while providing flexible access for analytics. This presents a difficult trade-off. Overly restrictive policies result in underutilization and data siloing, while insufficient restrictions can lead to privacy breaches and data leaks.

Differential privacy is widely recognized by experts as the most rigorous theoretical solution to this problem. Differential privacy provides a formal guarantee of privacy for individuals while allowing general statistical analysis of the data. Despite extensive academic research, differential privacy has not been widely adopted in practice. Additional work is needed to address performance and usability issues that arise when applying differential privacy to real-world environments.

In this dissertation we develop empirical and theoretical advances towards practical differential privacy. We conduct a study using 8.1 million real-world queries to determine the requirements for practical differential privacy, and identify limitations of previous approaches in light of these requirements. We then propose a novel method for differential privacy that addresses key limitations of previous approaches.

We present CHORUS, an open-source system that automatically enforces differential privacy for statistical SQL queries. CHORUS is the first system for differential privacy that is compatible with real databases, supports queries expressed in standard SQL, and integrates easily into existing data environments. Our evaluation demonstrates that CHORUS supports 93.9% of real-world statistical queries, integrates with production databases *without modifications to the database*, and scales to hundreds of millions of records. CHORUS is currently deployed at a large technology company for internal analytics and GDPR compliance. In this capacity, CHORUS processes more than 10,000 queries per day.

Dedicated to my mother Pat, a role model and inspiration for my graduate studies; and my wife Amina, a constant and dependable source of encouragement, love, and support. I am truly grateful to have both of you in my life.

Contents

Acknowledgments

I would like to thank my advisor Dawn Song, whose mentorship and support have been invaluable for my PhD studies and professional development. Going back further, I am extremely grateful for the opportunity you gave me as an undergraduate to work with your research group. This sparked my interest in academic research and inspired me to pursue a career in computer security. It has been quite a journey since then.

This thesis would not have been possible without the contributions of my fantastic research collaborators, in particular Joe Near and Joseph Hellerstein.

Finally, I would like to express my gratitude to the excellent faculty on the dissertation committee: Dawn Song, David Wagner, Deirdre Mulligan. Thank you for your helpful feedback, guidance and encouragement, from my qualifying exam to this dissertation.

This dissertation is based on work that appeared in the Proceedings of the VLDB Endowment [45]. The work was supported by the Center for Long-Term Cybersecurity, and DARPA & SPAWAR under contract N66001-15-C-4066. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Chapter 1

Introduction

As organizations increasingly collect sensitive information about individuals, these organizations are ethically and legally obligated to safeguard against privacy leaks. At the same time, they are motivated to extract maximum value by providing analysts with flexible access to the data. Unfortunately, allowing access to data has been a major cause of privacy breaches [12–14, 16].

This presents a difficult trade-off. Overly restrictive policies result in underutilization of data, while insufficient restrictions can lead to privacy breaches and data leaks. There is a growing and urgent need for solutions that can balance these interests by enabling general-purpose analytics while guaranteeing privacy protection for individuals.

Traditional approaches for data security are insufficient. Access control policies can limit access to a particular database, but once an analyst has access these policies cannot control how the data is used. Data anonymization attempts to provide privacy while allowing general-purpose analysis, but cannot be relied upon as demonstrated by a number of re-identification attacks [27, 58, 61, 70].

Differential privacy [29, 35] is widely recognized by experts as the most rigorous theoretical solution to this problem. Differential privacy provides a formal guarantee of privacy for individuals while allowing general statistical analysis of the data. In short, it states that the presence or absence of any single individual's data should not have a measurable effect on the results of a query. This allows precise answers to questions about populations in the data while guaranteeing the results reveal little about any individual. Unlike alternative approaches such as anonymization and k-anonymity, differential privacy protects against a wide range of attacks, including attacks using auxiliary information [27, 58, 61, 70].

Current research on differential privacy focuses on development of new algorithms, called *mechanisms*, to achieve differential privacy for a particular class of queries. These mechanisms work by adding random noise to results of the query; their principal goal is to provide high utility (i.e., low error) through judicious application of noise.

Researchers have developed dozens of mechanisms covering a broad range of use cases, from general-purpose statistical queries [20, 32, 54–56, 59, 63] to special-purpose analytics tasks such as graph analysis [23, 42, 46, 47, 66], range queries [18, 24, 41, 50–52, 64, 71–73], and analysis of data streams [33, 67]. Each mechanism works well for specific tasks and not as well, or not at all, on

other tasks.

Despite extensive academic research and an abundance of available mechanisms, differential privacy has not been widely adopted in practice. Existing applications of differential privacy in practice are limited to specialized use cases such as web browsing statistics [37] or keyboard and emoji use [15].

There are several major challenges for practical adoption of differential privacy. The first is seamless integration into real-world data environments. Data scientists often express analytics queries in SQL, an industry-standard query language supported by most major databases. A practical system for differential privacy should therefore provide robust support for SQL queries. A few mechanisms [20, 55–57, 59, 63] provide differential privacy for some subsets of SQL-like queries but none support the majority of queries in practice.

Additionally, real-world environments use highly customized data architectures and industrial-grade database engines individually tuned for performance and reliability. Previous differential privacy approaches require replacement of the database with a custom engine [55, 56, 63]. These approaches do not integrate easily into customized data architectures.

Another major challenge is simultaneously supporting different mechanisms. Current evidence suggests that there is no single "best mechanism" that performs optimally for all queries. Rather, the best mechanism depends on both the query and the dataset, and can also vary with the size of the dataset even for a single query [43]. A practical solution must therefore provide flexibility for mechanism selection and easy integration of new mechanisms. Previous differential privacy systems [45, 56, 63] implement only a single mechanism; these systems are not easily combined due to fundamental differences between their architectures.

Finally, although the theoretical aspects of differential privacy have been studied extensively, little is known about the quantitative impact of differential privacy on real-world queries. Recent work has evaluated this, but only for special-purpose analytics task such as histogram analysis [21] and range queries [43]. To the best of our knowledge, no existing work has explored the design and evaluation of differential privacy techniques for general, real-world queries.

In this dissertation we systematically address each of these challenges with the goal of supporting differential privacy in a practical setting. This dissertation describes three principal contributions towards this goal. First, we conduct the largest known empirical study of real-world SQL queries using a dataset of 8.1 million queries (Chapter 2). From these results we propose a new set of requirements for practical differential privacy on SQL queries.

To meet these requirements we propose elastic sensitivity (Chapter 3), a novel method for approximating the local sensitivity of queries with general equijoins. We prove that elastic sensitivity is an upper bound on local sensitivity and therefore can be used to enforce differential privacy using any local sensitivity-based mechanism.

We then present a novel approach in which *query rewriting* is used to enforce differential privacy for statistical SQL queries (Chapter 4). We demonstrate this approach using four general-purpose differential privacy mechanisms (including elastic sensitivity) and describe how additional mechanisms can be supported.

We develop CHORUS, an open-source system based on query rewriting that automatically enforces differential privacy for SQL queries. CHORUS is compatible with any SQL database that

supports standard math functions, requires no user modifications to the database or queries, and simultaneously supports multiple differential privacy mechanisms. To the best of our knowledge, no existing system provides these capabilities.

In the first evaluation of its kind, we use CHORUS to evaluate the four selected differential privacy mechanisms on real-world queries and data. The results demonstrate that our approach supports 93.9% of statistical queries in our corpus, integrates with a production DBMS *without any modifications to the database*, and scales to hundreds of millions of records.

CHORUS is currently deployed at Uber for its internal analytics. It represents a significant part of the company's GDPR compliance, and can provide both differential privacy and access control enforcement. In this capacity, CHORUS processes more than 10,000 queries per day.

We believe this dissertation represents a major first step towards addressing the challenges that have inhibited adoption of differential privacy in practice. A flexible, practical system supporting multiple state-of-the-art mechanisms and integrating easily into data environments could accelerate adoption of differential privacy in practice. In addition, the ability to evaluate current and future mechanisms in a real-world setting will support development of new mechanisms with greater utility and expand the application domain of differential privacy.

Chapter 2

Requirements for Practical Differential Privacy

2.1 Introduction

In this chapter we establish requirements for a practical differential privacy system using a dataset consisting of millions of real-world SQL queries. We then investigate the limitations of existing general-purpose differential mechanisms in light of these requirements.

Dataset. We use a dataset of SQL queries written by employees at Uber. The dataset contains 8.1 million queries executed between March 2013 and August 2016 on a broad range of sensitive data including rider and driver information, trip logs, and customer support data.

Data analysts at Uber query this information in support of many business interests such as improving service, detecting fraud, and understanding trends in the business. The majority of these use-cases require flexible, general-purpose analytics.

Given the size and diversity of our dataset, we believe it is representative of SQL queries in other real-world situations.

We investigate the following properties of queries in our dataset:

- **How many different database backends are used?** A practical differential privacy system must integrate with existing database infrastructure.
- **Which relational operators are used most frequently?** A practical differential privacy system must at a minimum support the most common relational operators.
- **What types of joins are used most frequently and how many are used in a typical query?** Making joins differentially private is challenging because the output of a join may contain duplicates of sensitive rows. This duplication is difficult to bound as it depends on the join type, join condition, and the underlying data. Understanding the different types of joins

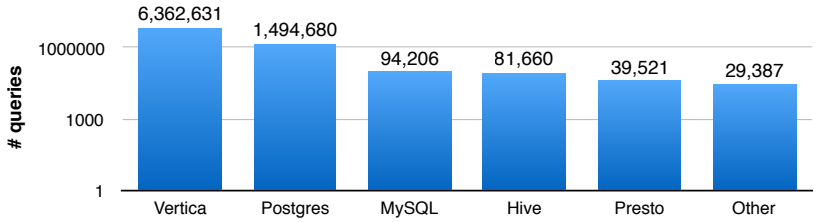
and their relative frequencies is therefore critical for supporting differential privacy on these queries.

- **What fraction of queries use aggregations and which aggregation functions are used most frequently?** Aggregation functions in SQL return statistics about populations in the data. Aggregation and non-aggregation queries represent fundamentally different privacy problems, as will be shown. A practical system must at minimum support the most common aggregations.
- **How complex are typical queries and how large are typical query results?** To be practical, a differential privacy mechanism must support real-world queries without imposing restrictions on query syntax, and it must scale to typical result sizes.

2.2 Study Results

We first summarize the study results, then define requirements of a practical differential privacy technique for real-world queries based on these results.

Question 1: How many different database backends are used?



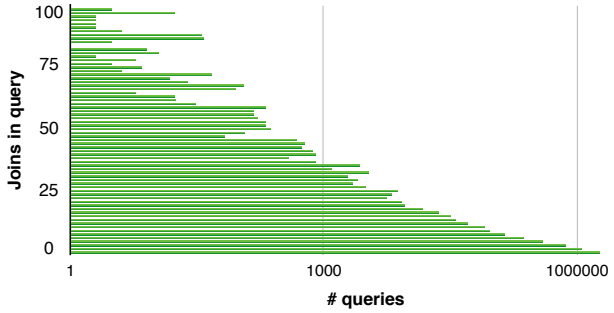
Results. The queries in our dataset use more than 6 database backends, including Vertica, Postgres, MySQL, Hive, and Presto.

Question 2: Which relational operators are used most frequently?

Operator	Frequency
`Select`	100%
`Join`	62.1%
`Union`	0.57%
`Minus/Except`	0.06%
`Intersect`	0.03%

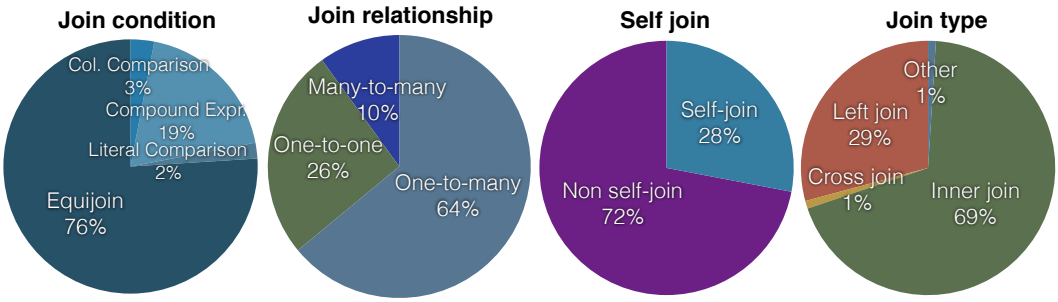
Results. All queries in our dataset use the `Select` operator, more than half of the queries use the `Join` operator, and fewer than 1 percent use other operators such as `Union`, `Minus`, and `Intersect`.

Question 3: How many joins are used by a typical query?



Results. A significant number of queries use multiple joins, with some queries using as many as 95 joins.

Question 4: What types of joins are used most frequently?



Join condition. The vast majority (76%) of joins are *equijoins*: joins that are conditioned on value equality of one column from both relations. A separate experiment (not shown) reveals that 65.9% of all join queries use *exclusively* equijoins.

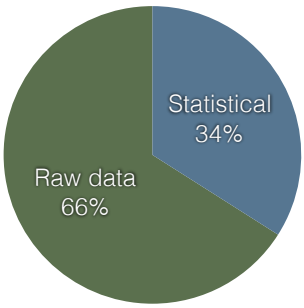
Compound expressions, defined as join conditions using function applications and conjunctions or disjunctions of primitive operators, account for 19% of join conditions. Column comparisons, defined as conditions that compare two columns using non-equality operators such as *greater than*, comprise 3% of join conditions. Literal comparisons, defined as join conditions comparing a single column to a string or integer literal, comprise 2% of join conditions.

Join relationship. A majority of joins (64%) are conditioned on one-to-many relationships, over one-quarter of joins (26%) are conditioned on one-to-one relationships, and 10% of joins are conditioned on many-to-many relationships.

Self join. 28% of queries include at least one self join, defined as a join in which the same database table appears in both joined relations. The remaining queries (72%) contain no self joins.

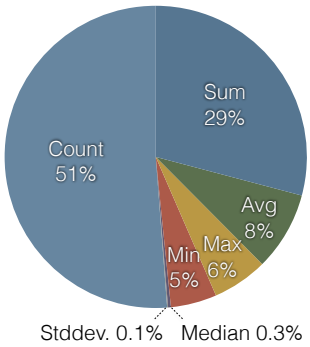
Join type. Inner join is the most common join type (69%), followed by left join (29%) and cross join (1%). The remaining types (right join and full join) together account for less than 1%.

Question 5: What fraction of queries use aggregations?



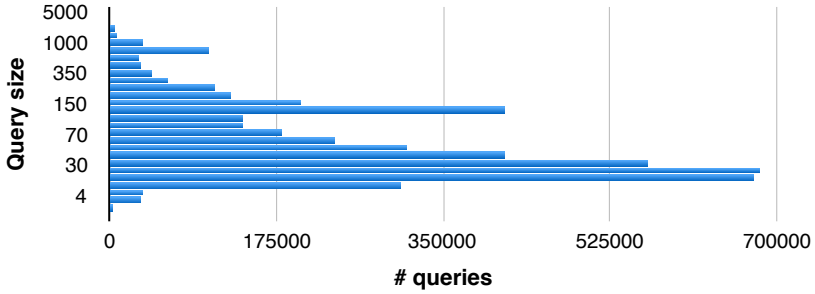
Results. Approximately one-third of queries are statistical, meaning they return only aggregations (count, average, etc.). The remaining queries return non-aggregated results (i.e., raw data) in at least one output column.

Question 6: Which aggregation functions are used most frequently?



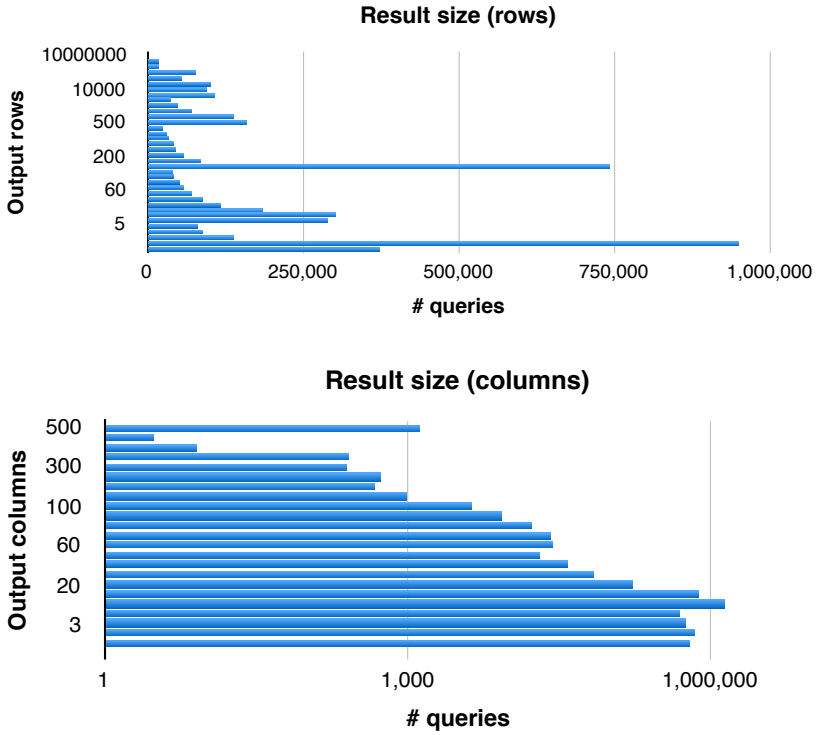
Results. `Count` is the most common aggregation function (51%), followed by `Sum` (29%), `Avg` (8%), `Max` (6%) and `Min` (5%). The remaining functions account for fewer than 1% of all aggregation functions.

Question 7: How complex are typical queries?



Results. The majority of queries are fewer than 100 clauses but a significant number of queries are much larger, with some queries containing as many as thousands of clauses.

Question 8: How large are typical query results?



Results. The output sizes of queries varies dramatically with respect to both rows and columns, and queries commonly return *hundreds* of columns and *hundreds of thousands* of rows.

2.3 Discussion of Results

Our study reveals that real-world queries are executed on many different database engines—in our dataset there are over 6. We believe this is typical; a variety of databases are commonly used within a company to match specific performance and scalability requirements. A practical mechanism for differential privacy will therefore allow the use of any of these existing databases, requiring neither a specific database distribution nor a custom execution engine in lieu of a standard database.

The study shows that 62.1% of all queries use SQL `Join`, and specifically equijoins which are by far the most common. Additionally, a majority of queries use multiple joins, more than one-quarter use self joins, and joins are conditioned on one-to-one, one-to-many, and many-to-many relationships. These results suggest that a practical differential privacy approach must at a minimum provide robust support for equijoins, including the full spectrum of join relationships and an arbitrary number of nested joins.

One-third (34%) of all queries return aggregate statistics. Differential privacy is principally designed for such queries, and in this dissertation we focus on these queries. Enforcing differential privacy for raw data queries is beyond the scope of this work, as differential privacy is generally not intended to address this problem.

For statistical queries, `Count` is by far the most common aggregation (39.3%). This validates the focus on counting and histogram queries by the majority of previous general-purpose differential privacy mechanisms [20, 55, 57, 63]. However, a significant percentage of queries (29%) use Sum, Avg, Min, and Max, suggesting that practical differential privacy system must additionally support these functions.

Requirements

We summarize our requirements for practical differential privacy of real-world SQL queries:

- **Requirement 1: Compatibility with existing databases.** A practical differential privacy approach must support heterogeneous database environments by not requiring a specific database distribution or replacement of the database with a custom runtime.
- **Requirement 2: Robust support for equijoin.** A practical differential privacy approach must provide robust support for equijoin, including both self joins and non-self joins, all join relationship types, and queries with an arbitrary number of nested joins.
- **Requirement 3: Support for standard aggregation functions.** A practical differential privacy approach must support the five standard SQL aggregation functions: `Count`, `Sum`, `Average`, `Max`, and `Min`, which represent 99% of aggregation functions used in real-world queries.

Our study shows that a differential privacy system satisfying these requirements is likely to have broad impact, supporting a majority of real-world statistical queries.

2.4 Existing Differential Privacy Mechanisms

Table 2.1 lists several existing differential privacy mechanisms and their supported features relative to our requirements. We summarize each below.

PINQ. Privacy Integrated Queries (PINQ) [55] is a mechanism that provides differential privacy for counting queries written in an augmented SQL dialect. PINQ supports a restricted join operator that groups together results with the same key. For one-to-one joins, this operator is equivalent to the standard semantics. For one-to-many and many-to-many joins, on the other hand, a PINQ query can count the number of unique join *keys* but not the number of joined *results*. Additionally, PINQ introduces new operators that do not exist in standard SQL, so the approach is not compatible with standard databases.

	Requirement 1	Requirement 2			Requirement 3
	Database compatability	1-to-1 equijoin	1-to-many equijoin	Many-many equijoin	Supported aggregations
PINQ [55]	no	✓			`Count`
wPINQ [63]	no	✓	✓	✓	`Count`
Sample & aggregate [56, 59]	no				`Avg, Sum, ...`
Restricted sensitivity [20]	no	✓	✓		`Count`
DJoin [57]	no	✓			`Count`

Table 2.1: Comparison of general-purpose differential privacy mechanisms.

wPINQ. Weighted PINQ (wPINQ) [63] extends PINQ with support for general equijoins and works by assigning a weight to each row in the database, then scaling down the weights of rows in a join to ensure an overall sensitivity of 1. In wPINQ, the result of a counting query is the sum of weights for records being counted plus noise drawn from the Laplace distribution. This approach allows wPINQ to support all three types of joins. However, wPINQ does not satisfy our database compatibility requirement. The system described by Proserpio et al. [63] uses a custom runtime; applying wPINQ in an existing database would require modifying the database to propagate weights during execution.

Sample & aggregate. Sample & aggregate [59] is a general-purpose mechanism that supports all statistical estimators (average, sum, min, max, etc.) and works by splitting the database into chunks, running the query on each chunk, and aggregating the results using a differentially private algorithm. Sample & aggregate cannot support joins, since splitting the database breaks join semantics.

Restricted sensitivity. Restricted sensitivity [20] is designed to bound the global sensitivity of counting queries with joins, by using properties of an auxiliary data model. The approach requires bounding the frequency of each join key globally (i.e. for all possible future databases). This works well for one-to-one and one-to-many joins, because the unique key on the "one" side of the join has a global bound. However, it cannot handle many-to-many joins, because the frequency of keys on *both* sides of the join may be unbounded. Blocki et al. [20] formalize the restricted sensitivity approach but do not describe how it could be used in a system compatible with existing databases, and no implementation is available.

DJoin. DJoin [57] is a mechanism designed for differentially private queries over datasets distributed over multiple parties. Due to the additional restrictions associated with this setting, DJoin supports only one-to-one joins, because it rewrites join queries as relational intersections. For example, consider the following query:

```
SELECT COUNT(*) FROM X JOIN Y ON X.A = Y.B
```

DJoin rewrites this query to the following (in relational algebra), which is semantically equivalent to the original query *only* if the join is one-to-one: $|\pi_A(X) \cap \pi_B(Y)|$. Additionally, the approach requires the use of special cryptographic functions during query execution, so it is not compatible with existing databases.

2.5 Summary

In summary we identified the following requirements for real-world differential privacy:

1. Compatibility with existing databases
2. Robust support for equijoin
3. Support for standard aggregation functions

We have identified the gaps of previous work based on these requirements. Specifically, no mechanism in Table 2.1 provides compatibility with existing databases and each mechanism satisfies (or partially satisfies) a different subset of Requirements 2 and 3.

In the remainder of this dissertation we develop new methods for satisfying all three requirements. We do this in two steps. First, in Chapter 3 we address Requirements 1 and 2 by proposing elastic sensitivity, a new differential privacy mechanism that is compatible with any existing database and supports general equijoins. Next in Chapter 4, we develop Chorus, a unified system that can simultaneously support multiple mechanisms (including elastic sensitivity), meeting all Requirements 1-3.

Chapter 3

Elastic Sensitivity

3.1 Introduction

This chapter proposes *elastic sensitivity*, a novel approach for differential privacy of SQL queries with equijoins. Elastic sensitivity was developed to meet Requirements 1 and 2 as described in Chapter 2.

In contrast to existing work, elastic sensitivity is compatible with real database systems, supports queries expressed in standard SQL, and integrates easily into existing data environments. For simplicity our presentation of elastic sensitivity focuses on counting queries. In Section 3.2 we discuss extensions to support other aggregation functions.

This chapter is organized as follows:

1. In Section 3.2 we introduce *elastic sensitivity*, a sound approximation of local sensitivity [31, 59] that supports general equijoins and can be calculated efficiently using only the query itself and a set of precomputed database metrics. We prove that elastic sensitivity is an upper bound on local sensitivity and can therefore be used to enforce differential privacy using any local sensitivity-based mechanism.

2. In Section 3.3 we design and implement FLEX, an end-to-end differential privacy system for SQL queries based on elastic sensitivity. We demonstrate that FLEX is compatible with any existing DBMS, can enforce differential privacy for the majority of real-world SQL queries, and incurs negligible (0.03%) performance overhead.

3. In Section 3.4 use FLEX to evaluate the impact of differential privacy on 9,862 real-world statistical queries in our dataset. In contrast to previous empirical evaluations of differential privacy, our experiment dataset contains a diverse variety of real-world queries executed on real data. We show that FLEX introduces low error for a majority of these queries.

3.2 Elastic Sensitivity

Elastic sensitivity is a novel approach for calculating an upper bound on a query's local sensitivity. After motivating the approach, we provide background on necessary concepts, formally define elastic sensitivity, and prove its correctness.

Motivation

Many previous differential privacy mechanisms [20, 55] are based on global sensitivity. These approaches do not generalize to queries with joins; the global sensitivity of queries with general joins may be *unbounded* because "a join has the ability to multiply input records, so that a single input record can influence an arbitrarily large number of output records." [55]

Techniques based on local sensitivity [31, 59] generally provide greater utility than global sensitivity-based approaches because they consider the *actual* database. Indeed, local sensitivity is finite for general queries with joins. However, directly computing local sensitivity is computationally infeasible, as it requires running the query on every possible neighbor of the true database—in our environment this would require running more than 1 billion queries for each original query. Previous work [59] describes efficient methods to calculate local sensitivity for a limited set of fixed queries (e.g., the median of all values in the database) but these techniques do not apply to general-purpose queries or queries with join.

These challenges are reflected in the design of previous mechanisms listed in Table 2.1. PINQ and restricted sensitivity support only joins for which global sensitivity can be bounded, and wPINQ scales weights attached to the data during joins to ensure a global sensitivity of 1. DJoin uses a measure of sensitivity unique to its distributed setting. None of these techniques is based on local sensitivity.

Elastic sensitivity is the first tractable approach to leverage local sensitivity for queries with general equijoins. The key insight of our approach is to model the impact of each join in the query using precomputed metrics about the frequency of join keys in the true database. This novel approach allows elastic sensitivity to compute a *conservative* approximation of local sensitivity without requiring any additional interactions with the database. In this section we prove elastic sensitivity is an upper bound on local sensitivity and can therefore be used with any local sensitivity-based differential privacy mechanism. In Section 3.3, we describe how to use elastic sensitivity to enforce differential privacy.

Background

We briefly summarize existing differential privacy concepts necessary for describing our approach. For a more thorough overview of differential privacy, we refer the reader to Dwork and Roth's excellent reference [35].

Differential privacy provides a formal guarantee of *indistinguishability*: a differentially private result does not yield very much information about which of two neighboring databases was used in calculating the result.

Formally, differential privacy considers a database modeled as a vector $x \in D^n$, in which x_i represents the data contributed by user i. The *distance* between two databases $x, y \in D^n$ is $d(x, y) = |\{i|x_i \neq y_i\}|$. Two databases x, y are *neighbors* if $d(x, y) = 1$.

Definition 1 (Differential privacy)**.** *A randomized mechanism $\mathcal{K} : D^n \to \mathbb{R}^d$ preserves (ϵ, δ)-differential privacy if for any pair of databases $x, y \in D^n$ such that $d(x, y) = 1$, and for all sets S of possible outputs:*

$$\textit{Pr}[\mathcal{K}(x) \in S] \leq e^{\epsilon}\textit{Pr}[\mathcal{K}(y) \in S] + \delta$$

Intuitively, the *sensitivity* of a query corresponds to the amount its results can change when the database changes. One measure of sensitivity is *global sensitivity*, which is the maximum difference in the query's result on *any* two neighboring databases.

Definition 2 (Global Sensitivity)**.** *For $f : D^n \to \mathbb{R}^d$ and all $x, y \in D^n$, the global sensitivity of f is*

$$GS_f = \max_{x,y:d(x,y)=1} ||f(x) - f(y)||$$

McSherry [55] defines the notion of *stable transformations* on a database, which we will use later. Intuitively, a transformation is stable if its privacy implications can be bounded.

Definition 3 (Global Stability)**.** *A transformation $T : D^n \to D^n$ is* c-stable *if for $x, y \in D^n$ such that $d(x, y) = 1$, $d(T(x), T(y)) \leq c$.*

Another definition of sensitivity is *local sensitivity* [31, 59], which is the maximum difference between the query's results on the *true database* and any neighbor of it:

Definition 4 (Local Sensitivity)**.** *For $f : D^n \to \mathbb{R}^d$ and $x \in D^n$, the local sensitivity of f at x is*

$$LS_f(x) = \max_{y:d(x,y)=1} ||f(x) - f(y)||$$

Local sensitivity is often much lower than global sensitivity since it is a property of the single true database rather than the set of all possible databases.

We extend the notion of stability to the case of local sensitivity by fixing x to be the true database.

Definition 5 (Local Stability)**.** *A transformation $T : D^n \to D^n$ is* locally c-stable *for true database x if for $y \in D^n$ such that $d(x, y) = 1$, $d(T(x), T(y)) \leq c$.*

Differential privacy for multi-table databases. In this chapter we consider *bounded* differential privacy [48], in which x can be obtained from its neighbor y by changing (but not adding or removing) a single tuple. Our setting involves a database represented as a multiset of tuples, and we wish to protect the presence or absence of a single tuple. If tuples are drawn from

the domain D and the database contains n tuples, the setting can be represented as a vector $x \in D^n$, in which $x_i = v$ if row i in the database contains the tuple v.

For queries without joins, a database $x \in D^n$ is considered as a single table. However, our setting considers database with multiple tables and queries with joins. We map this setting into the traditional definition of differential privacy by considering m tables $t_1, ..., t_m$ as disjoint subsets of a single database $x \in D^n$, so that $\bigcup_{i=1}^{m} t_i = x$.

With this mapping, differential privacy offers the same protection as in the single-table case: it protects the presence or absence of any single tuple in the database. When a single user contributes more than one protected tuple, however, protecting individual tuples may not be sufficient to provide privacy. Note that this caveat applies *equally* to the single- and multi-table cases—it is not a unique problem of multi-table differential privacy.

We maintain the same definition of neighboring databases as the single-table case. Neighbors of $x \in D^n$ can be obtained by selecting a table $t_i \in x$ and changing a single tuple, equivalent to changing a single tuple in a single-table database.

Smoothing functions. Because local sensitivity is based on the true database, it must be used carefully to avoid leaking information about the data. Prior work [31,59] describes techniques for using local sensitivity to enforce differential privacy. Henceforth we use the term *smoothing functions* to refer to these techniques. Smoothing functions are independent of the method used to compute local sensitivity, but generally require that local sensitivity can be computed an arbitrary distance k from the true database (i.e. when at most k entries are changed).

Definition 6 (Local Sensitivity at Distance)**.** *The local sensitivity of f at distance k from database x is:*

$$A_f^{(k)}(x) = \max_{y \in D^n : d(x,y) \leq k} LS_f(y)$$

Definition of Elastic Sensitivity

We define the *elastic sensitivity* of a query recursively on the query's structure. To allow the use of smoothing functions, our definition describes how to calculate elastic sensitivity at arbitrary distance k from the true database (under this definition, the local sensitivity of the query is defined at $k = 0$).

Figure 3.1 contains the complete definition, which is presented in four parts: (a) *Core relational algebra*, (b) *Definition of Elastic sensitivity*, (c) *Max frequency at distance k*, and (d) *Ancestors of a relation*. We describe each part next.

Core relational algebra. We present the formal definition of elastic sensitivity in terms of a subset of the standard relational algebra, defined in Figure 3.1(a). This subset includes selection (σ), projection (π), join ($\bowtie$), counting (*Count*), and counting with grouping ($\mathit{Count}_{G_1..G_n}$). It

admits arbitrary equijoins, including self joins, and all join relationships (one-to-one, one-to-many, and many-to-many).

To simplify the presentation our notation assumes the query performs a count as the *outermost* operation, however the approach naturally extends to aggregations nested anywhere in the query as long as the query does not perform arithmetic or other modifications to the aggregation result.

For example, the following query counts the total number of trips and projects the "count" attribute:

$$\pi_{\text{count}} \mathit{Count}(\text{trips})$$

Our approach can support this query by treating the inner relation as the query root.

Elastic sensitivity. Figure 3.1(b) contains the recursive definition of elastic sensitivity at distance k. We denote the elastic sensitivity of query q at distance k from the true database x as $\hat{S}^{(k)}(q, x)$. The $\hat{S}$ function is defined in terms of the elastic *stability* of relational transformations (denoted $\hat{S}_R$).

$\hat{S}^{(k)}_R(r, x)$ bounds the local stability (Definition 5) of relation r at distance k from the true database x. $\hat{S}^{(k)}_R(r, x)$ is defined in terms of $\mathsf{mf}_k(a, r, x)$, the maximum frequency of attribute a in relation r at distance k from database x.

Max frequency at distance k. The *maximum frequency* metric is used to bound the sensitivity of joins. We define the maximum frequency $\mathsf{mf}(a, r, x)$ as the frequency of the most frequent value of attribute a in relation r in the database instance x. In Section 3.3 we describe how the values of mf can be obtained from the database.

To bound the local sensitivity of a query at distance k from the true database, we must also bound the max frequency of each join key at distance k from the true database. For attribute a of relation r in the true database x, we denote this value $\mathsf{mf}_k(a, r, x)$, and define it (in terms of mf) in Figure 3.1(c).

Ancestors of a relation. The definition in Figure 3.1(d) is a formalization to identify self joins. Self joins have a much greater effect on sensitivity than joins of non-overlapping relations. In a self join, adding or removing one row of the underlying database may cause changes in *both* joined relations, rather than just one or the other. The join case of elastic sensitivity is therefore defined in two cases: one for self joins, and one for joins of non-overlapping relations. To distinguish the two cases, we use $\mathcal{A}(r)$ (defined in Figure 3.1(d)), which denotes the set of tables possibly contributing rows to r. A join of two relations r_1 and r_2 is a self join when r_1 and r_2 *overlap*, which occurs when some table t in the underlying database contributes rows to both r_1 and r_2. Relations r_1 and r_2 are non-overlapping when $|\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| = 0$.

Core relational algebra:

Attribute names	Relational transformations	Selection predicates	Counting queries
a	$R ::= t \mid R_1 \underset{x=y}{\bowtie} R_2$	$\varphi ::= a_1 \theta a_2 \mid a \theta v$	$Q ::= \mathit{Count}(R)$
Value constants	$\mid \Pi_{a_1,\ldots,a_n} R \mid \sigma_\varphi R$	$\theta ::= < \mid \leq \mid =$	$\mid \underset{G_1..G_n}{\mathit{Count}}(R)$
v	$\mid \mathit{Count}(R)$	$\mid \neq \mid \geq \mid >$	

(a)

Definition of elastic stability:

$$\hat{S}^{(k)}_R :: R \to D^n \to \textit{elastic stability}$$

$$\hat{S}^{(k)}_R(t,x) = 1$$

$$\hat{S}^{(k)}_R(r_1 \underset{a=b}{\bowtie} r_2, x) = \begin{cases} \max(\mathsf{mf}_k(a,r_1,x)\hat{S}^{(k)}_R(r_2,x), \mathsf{mf}_k(b,r_2,x)\hat{S}^{(k)}_R(r_1,x)) & |\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| = 0 \\ \mathsf{mf}_k(a,r_1,x)\hat{S}^{(k)}_R(r_2,x) + \mathsf{mf}_k(b,r_2,x)\hat{S}^{(k)}_R(r_1,x) + \hat{S}^{(k)}_R(r_1,x)\hat{S}^{(k)}_R(r_2,x) & |\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| > 0 \end{cases}$$

$$\hat{S}^{(k)}_R(\Pi_{a_1,\ldots,a_n} r, x) = \hat{S}^{(k)}_R(r,x)$$

$$\hat{S}^{(k)}_R(\sigma_\varphi r, x) = \hat{S}^{(k)}_R(r,x)$$

$$\hat{S}^{(k)}_R(\mathit{Count}(r)) = 1$$

Definition of elastic sensitivity:

$$\hat{S}^{(k)} :: Q \to D^n \to \textit{elastic sensitivity}$$

$$\hat{S}^{(k)}(\mathit{Count}(r), x) = \hat{S}^{(k)}_R(r,x)$$

$$\hat{S}^{(k)}(\underset{G_1..G_n}{\mathit{Count}}(r), x) = 2\hat{S}^{(k)}_R(r,x)$$

(b)

Maximum frequency at distance k:

$$\mathsf{mf}_k :: a \to R \to D^n \to \mathbb{N}$$

$$\mathsf{mf}_k(a,t,x) = \mathsf{mf}(a,t,x) + k$$

$$\mathsf{mf}_k(a_1, r_1 \underset{a_2=a_3}{\bowtie} r_2, x) = \begin{cases} \mathsf{mf}_k(a_1,r_1,x)\mathsf{mf}_k(a_3,r_2,x) & a_1 \in r_1 \\ \mathsf{mf}_k(a_1,r_2,x)\mathsf{mf}_k(a_2,r_1,x) & a_1 \in r_2 \end{cases}$$

$$\mathsf{mf}_k(a, \Pi_{a_1,\ldots,a_n} r, x) = \mathsf{mf}_k(a,r,x)$$

$$\mathsf{mf}_k(a, \sigma_\varphi r, x) = \mathsf{mf}_k(a,r,x)$$

$$\mathsf{mf}_k(a, \mathit{Count}(r), x) = \bot$$

(c)

Ancestors of a relation:

$$\mathcal{A} :: R \to \{R\}$$

$$\mathcal{A}(t) = \{t\}$$

$$\mathcal{A}(r_1 \underset{a=b}{\bowtie} r_2) = \mathcal{A}(r_1) \cup \mathcal{A}(r_2)$$

$$\mathcal{A}(\Pi_{a_1,\ldots,a_n} r) = \mathcal{A}(r)$$

$$\mathcal{A}(\sigma_\varphi r) = \mathcal{A}(r)$$

(d)

Figure 3.1: (a) syntax of core relational algebra; (b) definition of elastic stability and elastic sensitivity at distance k; (c) definition of maximum frequency at distance k; (d) definition of ancestors of a relation.

Join conditions. For simplicity our notation refers only to the case where a join contains a single equality predicate. The approach naturally extends to join conditions containing *any* predicate that can be decomposed into a conjunction of an equijoin term and any other terms. Consider for example the following query:

```
SELECT COUNT(*) FROM a
JOIN b ON a.id = b.id AND a.size > b.size
```

Calculation of elastic sensitivity for this query requires only the equijoin term ($a.id = b.id$) and therefore follows directly from our definition. Note that in a conjunction, each predicate adds additional constraints that may decrease (but never increase) the true local stability of the join, hence our definition correctly computes an upper bound on the stability.

Example: Counting Triangles

We now consider step-by-step calculation of elastic sensitivity for an example query. We select the problem of counting triangles in a directed graph, described by Prosperio et al. in their evaluation of WPINQ [63]. This example contains multiple self-joins, which demonstrate the most complex recursive cases of Figure 3.1.

Following Prosperio et al. we select privacy budget $\epsilon = 0.7$ and consider the `ca-HepTh` [8] dataset, which has maximum frequency metric of 65.

In SQL, the query is expressed as:

```
SELECT COUNT(*) FROM edges e1
JOIN edges e2 ON e1.dest = e2.source AND
                 e1.source < e2.source
JOIN edges e3 ON e2.dest = e3.source AND
                 e3.dest = e1.source AND
                 e2.source < e3.source
```

Consider the first join ($e_1 \bowtie e_2$), which joins the `edges` table with itself. By definition of $\hat{S}^{(k)}_R$ (self join case) the elastic stability of this relation is:

$$\mathsf{mf}_k(\mathit{dest}, \texttt{edges}, x)\hat{S}^{(k)}_R(\texttt{edges}, x) + \ \mathsf{mf}_k(\mathit{source}, \texttt{edges}, x)\hat{S}^{(k)}_R(\texttt{edges}, x) + \ \hat{S}^{(k)}_R(\texttt{edges}, x)\hat{S}^{(k)}_R(\texttt{edges}, x)$$

Furthermore, since `edges` is a table, $\hat{S}^{(k)}_R(\texttt{edges}) = 1$.

We then have:

$$\mathsf{mf}_k(\mathit{dest}, \texttt{edges}, x) = \mathsf{mf}(\mathit{dest}, \texttt{edges}, x) + k$$
$$\mathsf{mf}_k(\mathit{source}, \texttt{edges}, x) = \mathsf{mf}(\mathit{source}, \texttt{edges}, x) + k$$

Substituting the max frequency metric (65), the elastic stability of this relation is:

$$(65 + k) + (65 + k) + 1 = 131 + 2k.$$

Now consider the second join, which joins e_3 (an alias for the `edges` table) with the previous joined relation ($e_1 \bowtie e_2$). Following the same process and substituting values, the elastic stability of this relation is:

$$\begin{aligned}
&\mathsf{mf}_k(\textit{dest}, \texttt{edges}, x)\hat{S}^{(k)}_R(e_1 \bowtie e_2, x) + \mathsf{mf}_k(\textit{source}, \texttt{edges}, x)\hat{S}^{(k)}_R(\texttt{edges}, x) + \hat{S}^{(k)}_R(e_1 \bowtie e_2, x)\hat{S}^{(k)}_R(\texttt{edges}, x)\\
=&(65+k)(131+2k) + (65+k) + (131+2k)\\
=&2k^2 + 199k + 8711
\end{aligned}$$

This expression describes the elastic stability at distance k of relation $(e_1 \bowtie e_2) \bowtie e_3$. Per the definition of $\hat{S}^{(k)}$ the elastic sensitivity of a counting query is equal to the elastic stability of the relation being counted, therefore this expression defines the elastic sensitivity of the full original query.

As we will discuss in Section 3.3, elastic sensitivity must be smoothed using smooth sensitivity [59] before it can be used with the Laplace mechanism. In short, this process requires computing the maximum value of elastic sensitivity at k multiplied by an exponentially decaying function in k:

$$\begin{aligned}
S &= \max_{k=0,1,\ldots,n} e^{-\beta k}\hat{S}^{(k)}\\
&= \max_{k=0,1,\ldots,n} e^{-\beta k}(2k^2 + 199k + 8711)
\end{aligned}$$

where $\beta = \frac{\epsilon}{2\ln(2/\delta)}$ and $\delta = 10^{-8}$.

The maximum value is $S = 8896.95$, which occurs at distance $k = 19$. Therefore, to enforce differential privacy we add Laplace noise scaled to $\frac{2S}{\epsilon} = \frac{17793.9}{0.7}$, per Definition 7 (see Section 3.3).

Elastic Sensitivity is an Upper Bound on Local Sensitivity

In this section, we prove that elastic sensitivity is an upper bound on the local sensitivity of a query. This fundamental result affirms the soundness of using elastic sensitivity in any local sensitivity-based differential privacy mechanism.

First, we prove two important lemmas: one showing the correctness of the max frequency at distance k, and the other showing the correctness of elastic stability.

Lemma 1. *For database x, at distance k, r has at most $\mathsf{mf}_k(a, r, x)$ occurrences of the most popular join key in attribute a:*

$$\mathsf{mf}_k(a, r, x) \geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a, r, y)$$

Proof. By induction on the structure of r.

Case t. To obtain the largest possible number of occurrences of the most popular join key in a table t at distance k, we modify k rows to contain the most popular join key. Thus, $\max_{y:d(x,y)\leq k} \mathsf{mf}(a, r, y) = \mathsf{mf}(a, r, x) + k$.

Case $r_1 \underset{a_2=a_3}{\bowtie} r_2$. We need to show that:

$$\mathsf{mf}_k(a_1, r_1 \underset{a_2=a_3}{\bowtie} r_2, x) \geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a_1, r_1 \underset{a_2=a_3}{\bowtie} r_2, y) \tag{3.1}$$

Consider the case when $a_1 \in r_1$ (the proof for case $a_1 \in r_2$ is symmetric). The worst-case sensitivity occurs when each tuple in r_1 with the most popular value for a_1 also contains attribute value a_2 matching the most popular value of attribute a_3 in r_2. So we can rewrite equation 3.1:

$$\mathsf{mf}_k(a_1, r_1 \underset{a_2=a_3}{\bowtie} r_2, x) \geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a_1, r_1, y)\mathsf{mf}(a_3, r_2, y) \tag{3.2}$$

We then rewrite the left-hand side, based on the definition of mf_k and the inductive hypothesis. Each step may make the left-hand side smaller, but never larger, preserving the original inequality:

$$\begin{aligned}
&\mathsf{mf}_k(a_1, r_1 \underset{a_2=a_3}{\bowtie} r_2, x) \\
&= \mathsf{mf}_k(a_1, r_1, x)\mathsf{mf}_k(a_3, r_2, x) \\
&\geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a_1, r_1, y) \max_{y:d(x,y)\leq k} \mathsf{mf}(a_3, r_2, y) \\
&\geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a_1, r_1, y)\mathsf{mf}(a_3, r_2, y)
\end{aligned}$$

Which is equal to the right-hand side of equation 3.2.

Case $\Pi_{a_1,\ldots,a_n} r$. Projection does not change the number of rows, so the conclusion follows directly from the inductive hypothesis.

Case $\sigma_\varphi r$. Selection might filter out some rows, but does not modify attribute values. In the worst case, no rows are filtered out, so $\sigma_\varphi r$ has the same number of occurrences of the most popular join key as r. The conclusion thus follows directly from the inductive hypothesis. □

Lemma 2. *$\hat{S}^{(k)}_R(r)$ is an upper bound on the local stability of relation expression r at distance k from database x:*

$$A^{(k)}_{\mathit{count}(r)}(x) \leq \hat{S}^{(k)}_R(r, x)$$

Proof. By induction on the structure of r.

Case t. The stability of a table is 1, no matter its contents.

Case $r_1 \underset{a=b}{\bowtie} r_2$. We want to bound the number of changed rows in the joined relation. There are two cases, depending on whether or not the join is a self join.

Subcase 1: no self join. When the ancestors of r_1 and r_2 are non-overlapping (i.e. $|\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| = 0$), then the join is not a self join. This means that either r_1 may change or r_2 may change, *but not both*. As a result, either $\hat{S}^{(k)}_R(r_1, x) = 0$ or $\hat{S}^{(k)}_R(r_2, x) = 0$. We therefore have two cases:

1. When $\hat{S}^{(k)}_R(r_1, x) = 0$, r_2 may contain at most $\hat{S}^{(k)}_R(r_2, x)$ changed rows, producing at most $\mathsf{mf}_k(a, r_1, x)\hat{S}^{(k)}_R(r_2, x)$ changed rows in the joined relation.

2. In the symmetric case, when $\hat{S}^{(k)}_R(r_2, x) = 0$, the joined relation contains at most $\mathsf{mf}_k(b, r_2, x)\hat{S}^{(k)}_R(r_1, x)$ changed rows.

We choose to modify the relation resulting in the largest number of changed rows, which is exactly the definition.

Subcase 2: self join. When the set of ancestor tables of r_1 overlaps with the set of ancestor tables of r_2, i.e. $|\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| > 0$, then changing a single row in the database could result in changed rows in *both* r_1 and r_2.

In the self join case, there are three sources of changed rows:

1. The join key of an original row from r_1 could match the join key of a changed row in r_2.
2. The join key of an original row from r_2 could match the join key of a changed row in r_1.
3. The join key of a changed row from r_1 could match the join key of a changed row in r_2.

Now consider how many changed rows could exist in each class.

1. In class 1, r_2 could have at most $\hat{S}^{(k)}_R(r_2, x)$ changed rows (by the inductive hypothesis). In the worst case, each of these changed rows matches the *most popular* join key in r_1, which occurs at most $\mathsf{mf}_k(a, r_1, x)$ times (by Lemma 1), so class 1 contains at most $\mathsf{mf}_k(a, r_1, x)\hat{S}^{(k)}_R(r_2, x)$ changed rows.
2. Class 2 is the symmetric case of class 1, and thus contains at most $\mathsf{mf}_k(b, r_2, x)\hat{S}^{(k)}_R(r_1, x)$ changed rows.
3. In class 3, we know that r_1 contains at most $\hat{S}^{(k)}_R(r_1, x)$ changed rows and r_2 contains at most $\hat{S}^{(k)}_R(r_2, x)$ changed rows. In the worst case, all of these changed rows contain the same join key, and so the joined relation contains $\hat{S}^{(k)}_R(r_1, x)\hat{S}^{(k)}_R(r_2, x)$ changed rows.

The total number of changed rows is therefore bounded by the sum of the bounds on the three classes:

$$\mathsf{mf}_k(a, r_1, x)\hat{S}^{(k)}_R(r_2, x) + \mathsf{mf}_k(b, r_2, x)\hat{S}^{(k)}_R(r_1, x) + \hat{S}^{(k)}_R(r_1, x) \qquad \hat{S}^{(k)}_R(r_2, x)$$

Which is exactly the definition.

Case $\Pi_{a_1,\ldots,a_n} r$. Projection does not change rows. The conclusion therefore follows from the inductive hypothesis.

Case $\sigma_\varphi r$. Selection does not change rows. The conclusion therefore follows from the inductive hypothesis.

Case *Count*(r). Count without grouping produces a relation with a single row. The stability of such a relation is 1, at any distance.

□

Main Theorem

We are now prepared to prove the main theorem.

Theorem 1. *The elastic sensitivity $\hat{S}^{(k)}(q, x)$ of a query q at distance k from the true database x is an upper bound on the local sensitivity $A_q^{(k)}(x)$ of q executed at distance k from database x:*

$$A_q^{(k)}(x) \leq \hat{S}^{(k)}(q, x)$$

Proof. There are two cases: histogram queries and non-histogram queries.

Case *Count*(r) (non-histogram). The local sensitivity of a non-histogram counting query over r is equal to the stability of r, so the result follows directly from Lemma 2.

Case *Count*$_{G_1..G_n}(r)$ (histogram). In a histogram query, each changed row in the underlying relation can change two rows in the histogram [32]. Thus by Lemma 2, the histogram's local stability is bounded by $2\hat{S}_R^{(k)}(r, x)$. □

Optimization for Public Tables

Our definition of elastic sensitivity assumes that all database records must be protected. In practice, databases often contain a mixture of sensitive and non-sensitive data. This fact can be used to tighten our bound on local sensitivity for queries joining on non-sensitive tables.

In our dataset, for example, city data is publicly known, therefore the system does not need to protect against an attacker learning information about the cities table. Note the set of public tables is domain-specific and will vary in each data environment.

More precisely, in a join expression `T1 JOIN T2 ON T1.A = T2.B`, if `T2` is publicly known, the elastic stability of the join is equal to the elastic stability of `T1` times the maximum frequency of `T2.B`. This formulation prevents the use of a publicly-known table with repeated join keys from revealing information about a private table.

Discussion of Limitations and Extensions

This section discusses limitations of elastic sensitivity and potential extensions to support other common aggregation functions.

Unsupported Queries

Elastic sensitivity does not support non-equijoins, and adding support for these is not straightforward. Consider the query:

```
SELECT count(*) FROM A JOIN B ON A.x > B.y
```

This query compares join keys using the greater-than operator, and bounding the number of matches for this comparison would require knowledge about *all* the data for `A.x` and `B.y`.

Fortunately, as demonstrated in Chapter 2, more than three-quarters of joins are equijoins. Elastic sensitivity could be extended to support other join types by querying the database for necessary data-dependent bounds, but this modification would require interactions with the database for each original query.

Elastic sensitivity can also fail when requisite max-frequency metrics are not available due to the query structure. Consider the query:

```
WITH A AS (SELECT count(*) FROM T1),
     B AS (SELECT count(*) FROM T2)
SELECT count(*) FROM A JOIN B ON A.count = B.count
```

This query uses counts computed in subqueries as join keys. Because the mf metric covers only the attributes available in the original tables of the database, our approach cannot bound the sensitivity of this query and must reject it. In general, elastic sensitivity applies only when join keys are drawn directly from original tables. Fortunately, this criterion holds for 98.5% of joins in our dataset, so this limitation has very little consequence in practice.

Supporting Other Aggregation Functions

In this section we outline extensions of our approach to support non-count aggregation functions, and characterize the expected utility for each. These extensions, which provide a roadmap for potential future research, would expand the set of queries supported by an elastic sensitivity-based system.

Value range metric. To describe these extensions we define a new metric, *value range* $\mathsf{vr}(a, r)$, defined as the maximum value minus the minimum value allowed by the data model of column a in relation r.

This metric can be derived in a few ways. First, it can be extracted automatically from the database's column constraint definitions [2], if they exist. Second, a SQL query can extract the *current* value range, which can provide a guideline for selecting the permissible value range based on records already in the database; finally, a domain expert can define the metric using knowledge about the data's semantics.

Once the metric is defined, it must be enforced in order for differential privacy to be guaranteed. The metric could be enforced as a data integrity check, for example using column check constraints [2].

Sum and Average. For sum and average, we note that the local sensitivity of these functions is affected both by the stability of the underlying relation, because each row of the relation potentially contributes to the computed sum or average, and by the range of possible values of the attributes involved.

Given our definition of vr above, the elastic sensitivity of both `Sum` and `Average` on relation r at distance k from database x is defined by $\mathsf{vr}(a, r)S^{(k)}_R(r, x)$. This expression captures the largest possible change in local sensitivity, in which each new row in r has the maximum value of a, for a total change of $\mathsf{vr}(a, r)$ per row.

For *Sum* queries on relations with stability 1 (i.e. relations without joins), this definition of elastic sensitivity is exactly equal to the query's local sensitivity, so the approach will provide optimal utility. As the relation's stability grows, so does the gap between elastic sensitivity and local sensitivity, and utility degrades, since elastic sensitivity makes the worst-case assumption that each row duplicated by a join contains the maximum value allowed by the data model.

For the *average* function, this definition is exactly equal to local sensitivity only for the degenerate case of averages of a single row. As more input rows are added, local sensitivity shrinks, since the impact of a single new row is amortized over the number of averaged records, while elastic sensitivity remains constant. Therefore utility degradation is proportional to both the stability of the relation as well as the number of records being averaged.

This could be mitigated with a separate analysis to compute a *lower bound* on the number of records being averaged, in which case the sensitivity could be scaled down by this factor. Such an analysis would require inspection of filter conditions in the query and an expanded set of database metrics.

Max and min. We observe that the stability of the underlying relation has no effect on the local sensitivity of *max* and *min*. Consequently, for such queries the data model $\mathsf{vr}(a, r)$ directly provides the *global sensitivity*, which is an upper bound of local sensitivity. However, the max and min functions are inherently sensitive, because they are strongly affected by outliers in the database [31], and therefore *any* differential privacy technique will provide poor utility in the general case.

Due to this fundamental limitation, previous work [31, 59, 68] has focused on the *robust* counterparts of these functions, such as the interquartile range, which are less sensitive to changes in the database. This strategy is not viable in our setting since functions like interquartile range are not supported by standard SQL.

3.3 FLEX: Differential Privacy via Elastic Sensitivity

This section describes FLEX, our system to enforce differential privacy for SQL queries using elastic sensitivity. Figure 3.2 summarizes the architecture of our system. For a given SQL query, FLEX uses an analysis of the query to calculate its elastic sensitivity, as described in Section 3.2. FLEX then applies smooth sensitivity [59, 60] to the elastic sensitivity and finally adds noise drawn from the Laplace distribution to the original query results. In Section 3.3 we prove this approach provides (ϵ, δ)-differential privacy.

Importantly, our approach allows the query to execute on any existing database. FLEX requires only static analysis of the query and post-processing of the query results, and requires no inter-

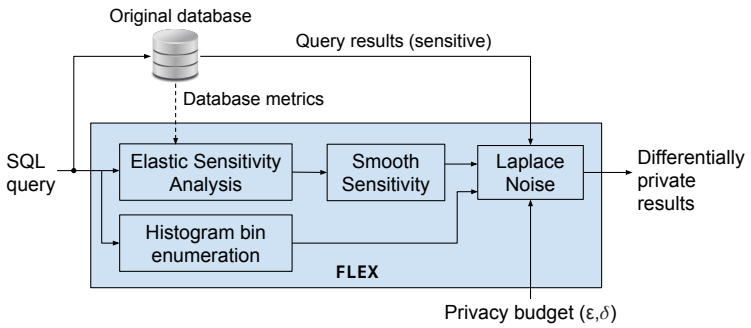


Figure 3.2: Architecture of FLEX.

actions with the database to enforce differential privacy. As we demonstrate in Section 3.4, this design allows the approach to scale to big data while incurring minimal performance overhead.

Collecting max frequency metrics. The definition of elastic sensitivity requires a set of precomputed metrics mf from the database, defined as the frequency of the most frequent attribute for each join key. The values of mf can be easily obtained with a SQL query. For example, this query retrieves the metric for column a of table T:

```
SELECT COUNT(a) FROM T GROUP BY a
ORDER BY count DESC LIMIT 1;
```

Obtaining these metrics is a separate step from enforcing differential privacy for a query; the metrics can be obtained once and re-used for all queries. Note the metric must be recomputed when the most frequent join attribute changes, otherwise differential privacy is no longer guaranteed. For this reason, the architecture in Figure 3.2 is ideal for environments where database updates are far less frequent than database queries.

Most databases can be configured using triggers [4] to automatically recompute the metrics on database updates; this approach could support environments with frequent data updates.

Elastic Sensitivity analysis. To compute elastic sensitivity we built an analysis framework for SQL queries based on the Presto parser [9], with additional logic to resolve aliases and a framework to perform abstract interpretation-based dataflow analyses on the query tree. FLEX's elastic sensitivity analysis is built on this dataflow analysis engine, and propagates information about ancestor relations and max-frequency metrics for each joined column in order to compute the overall elastic sensitivity of the query, per the recursive definition in Section 3.2. We evaluate the execution time and success rate of this analysis in Section 3.4.

Histogram bin enumeration. When a query uses SQL's `GROUP BY` construct, the output is a histogram containing a set of bin labels and an aggregation result (e.g., count) for each bin. To simplify presentation, our definition of elastic sensitivity in Section 3.2 assumes that the analyst provides the desired histogram bins labels ℓ. This requirement, also adopted by previous work [55],

is necessary to prevent leaking information via the presence or absence of a bin. In practice, however, analysts do not expect to provide histogram bin labels manually.

In some cases, FLEX can automatically build the set of histogram bin labels ℓ for a given query. In our dataset, many histogram queries use non-protected bin labels drawn from finite domains (e.g. city names or product types). For each possible value of the histogram bin label, FLEX can automatically build ℓ and obtain the corresponding differentially private count for that histogram bin. Then, FLEX adds a row to the output containing the bin label and its differentially private count, where results for missing bins are assigned value 0 and noise is added as usual.

This process returns a histogram of the expected form which does not reveal anything new through the presence or absence of a bin. Additionally, since this process requires the bin labels to be non-protected, the original bin labels can be returned. The process can generalize to any aggregation function.

The above approach requires a finite, enumerable, and non-protected set of values for each histogram bin label. When the requirement cannot be met, for example because the histogram bin labels are protected or cannot be enumerated, FLEX can still return the differentially private count for each bin, but it must rely on the analyst to specify the bin labels.

Proof of Correctness

In this section we formally define the FLEX mechanism and prove that it provides (ϵ, δ)-differential privacy.

FLEX implements the following differential privacy mechanism derived from the Laplace-based smooth sensitivity mechanism defined by Nissim et al. [59, 60]:

Definition 7 (FLEX mechanism)**.** *For input query q and histogram bin labels ℓ on true database x of size n, with privacy parameters (ϵ, δ):*

1. *Set* $\beta = \frac{\epsilon}{2\ln(2/\delta)}$.
2. *Calculate* $S = \max_{k=0,1,\ldots,n} e^{-\beta k} \hat{S}^{(k)}(q, x)$.
3. *Release* $q_\ell(x) + \mathit{Lap}(2S/\epsilon)$.

This mechanism leverages smooth sensitivity [59, 60], using elastic sensitivity as an upper bound on local sensitivity.

Theorem 2. *The* FLEX *mechanism provides (ϵ, δ)-differential privacy.*

Proof. By Theorem 1 and Nissim et al. [60] Lemma 2.3, S is a β-smooth upper bound on the local sensitivity of q. By Nissim et al. Lemma 2.9, when the Laplace mechanism is used, a setting of $\beta = \frac{\epsilon}{2\ln(2/\delta)}$ suffices to provide (ϵ, δ)-differential privacy. By Nissim et al. Corollary 2.4, the value released by the FLEX mechanism is (ϵ, δ)-differentially private. □

Efficiently Calculating S

The definition of the FLEX mechanism (Definition 7) requires calculating the maximum smooth sensitivity over all distances k between 0 and n (the size of the true database). For large databases as in our setting, this can require hundred of millions of calculations for each query. Even if each calculation is fast, this can be highly inefficient.

Fortunately, the particular combination of elastic sensitivity with smooth sensitivity allows for an optimization. The elastic sensitivity $\hat{S}^{(k)}(q,x)$ grows as $k^{j(q)^2}$, where $j(q)$ is the number of joins in q (see Lemma 3 below). For a given query, $j(q)$ is fixed. The smoothing factor ($e^{-\beta k}$), on the other hand, shrinks exponentially in k.

Recall that the smoothed-out elastic sensitivity at k is $S(k) = e^{-\beta k}\hat{S}^{(k)}(q,x)$. We will show that to find $\max_{k=0,1,\ldots,n} S(k)$, it is sufficient to find $\max_{k=0,1,\ldots,m} S(k)$, where $m \geq \frac{j(q)^2}{\beta}$. Since m is typically much smaller than n (and depends on the query, rather than the size of the database), this yields significant computational savings.

Lemma 3. *For all relation expressions r and databases x, where $j(r)$ is the number of joins in r, $\hat{S}^{(k)}_R(r,x)$ is a polynomial in k of degree at most $j(r)^2$, and all coefficients are non-negative.*

Proof. Follows from the recursive definitions of $\hat{S}^{(k)}_R$ and mf_k, since each makes at most $j(r)$ recursive calls and only adds or multiplies the results. □

Theorem 3. *For all queries q and databases x, the smoothed-out elastic sensitivity at distance k is $S(k) = e^{-\beta k}\hat{S}^{(k)}(q,x)$. For each x and q, if q queries a relation r, the maximum value of $S(k)$ occurs from $k = 0$ to $k = \frac{j(r)^2}{\beta}$.*

Proof. Let the constant $\lambda = j(r)^2$. By Lemma 3, we have that for some set of constants $\alpha_0, \ldots \alpha_\lambda$:

$$\hat{S}^{(k)}_R(r,x) = \sum_{i=0}^{\lambda} \alpha_i k^{\lambda - i}$$

We therefore have that:

$$S(k) = \frac{\sum_{i=0}^{\lambda} \alpha_i k^{\lambda-i}}{e^{\beta k}}$$

$$S'(k) = \sum_{i=0}^{\lambda} \alpha_i e^{\beta k} k^{\lambda-i-1}(\lambda - i - \beta k)$$

Under the condition that $\alpha_i \geq 0$, each term in the numerator is ≤ 0 exactly when $\lambda - i - \beta k \leq 0$. We know that $\alpha_i \geq 0$ by the definition of elastic sensitivity.

We also know that $\lambda \geq 0$, because a query cannot have a negative number of joins. Thus the first term ($i = 0$) is ≤ 0 exactly when $k \geq \frac{\lambda}{\beta}$ (we know that $\beta \geq 0$ by its definition). All of the other terms will also be ≤ 0 when $k \geq \frac{\lambda}{\beta}$, because for $i > 0$, $\lambda - i - \beta k < \lambda - \beta k$.

We can therefore conclude that $S'(k) \leq 0$ when $k > \frac{\lambda}{\beta}$, and so $S(k)$ is flat or decreasing for $k > \frac{j(r)^2}{\beta}$. □

Privacy Budget & Multiple Queries

FLEX does not prescribe a specific privacy budget management strategy, allowing the use existing privacy budget methods as needed for specific applications. Below we provide a brief overview of several approaches.

Composition techniques. Composition for differential privacy [32] provides a simple way to support multiple queries: the ϵs and δs for these queries simply add up until they reach a maximum allowable budget, at which point the system refuses to answer new queries. The *strong composition theorem* [36] improves on this method to produce a tighter bound on the privacy budget used. Both approaches are independent of the mechanism and therefore apply directly to FLEX.

Budget-efficient approaches. Several approaches answer multiple queries together (i.e. in a single workload) resulting in more efficient use of a given privacy budget than simple composition techniques. These approaches work by posing counting queries through a low-level differentially private interface to the database. FLEX can provide the low-level interface to support these approaches.

The *sparse vector technique* [34] answers only queries whose results lie above a predefined threshold. This approach depletes the privacy budget for answered queries only. The *Matrix Mechanism* [51] and *MWEM* [41] algorithms build an approximation of the true database using differentially private results from the underlying mechanism; the approximated database is then used to answer questions in the workload. Ding et al. [28] use a similar approach to release differentially private data cubes. Each of these mechanisms is defined in terms of the Laplace mechanism and thus can be implemented using FLEX.

3.4 Experimental Evaluation

We evaluate our approach with the following experiments:

- We measure the performance overhead and success rate of FLEX on real-world queries.
- We investigate the utility of FLEX-based differential privacy for real-world queries with and without joins.
- We evaluate the effect of the privacy budget ϵ on the utility of FLEX-based differential privacy.
- We measure the utility impact of the public table optimization described in Section 3.2.
- We compare FLEX and wPINQ on a set of representative counting queries using join.

	Avg (s)	Max (s)
Original query	42.4	3,452
FLEX*: Elastic Sensitivity Analysis*	0.007	1.2
FLEX*: Output Perturbation*	0.005	2.4

Table 3.1: Performance of FLEX-based differential privacy.

Experimental setup & dataset. We ran all of our experiments using our implementation of FLEX with Java 8 on Mac OSX. Our test machine was equipped with a 2.2 GHz Intel Core i7 and 8GB of memory. Our experiment dataset includes 9,862 real queries executed during October 2016. To build this dataset, we identified all counting queries submitted during this time period that examined sensitive trip data. Our dataset also includes the original (non-differentially private) results of each query.

Success Rate and Performance of FLEX

To investigate FLEX's support for the wide range of SQL features in real-world queries, we ran FLEX's elastic sensitivity analysis on the queries in our experiment dataset. We recorded the number of errors and classified each error according to its type.

In total, FLEX successfully calculated elastic sensitivity for 76% of the queries. The largest group of errors is due to unsupported queries (14.14%). These queries use features for which our approach cannot compute an elastic sensitivity, as described in Section 3.2. Parsing errors occurred for 6.58% of queries. These errors result from incomplete grammar definitions for the full set of SQL dialects used by the queries, and could be fixed by expanding Presto parser's grammar definitions. The remaining errors (3.21%) are due to other causes.

To investigate the performance of FLEX-based differential privacy, we measured the total execution time of the architecture described in Figure 3.2 compared with the original query execution time. We report the results in Table 3.1. Parsing and analysis of the query to calculate elastic sensitivity took an average of 7.03 milliseconds per query. The output perturbation step added an additional 4.86 milliseconds per query. By contrast, the average database execution time was 42.4 seconds per query, implying an average performance overhead of 0.03%.

Utility of FLEX on Real-World Queries

Our work is the first to evaluate differential privacy on a set of real-world queries. In contrast with previous evaluations of differential privacy [21, 43, 44], our dataset includes a wide variety of real queries executed on real data.

We evaluate the behavior of FLEX for this broad range of queries. Specifically, we measure the noise introduced to query results based on whether or not the query uses join and what percentage of the data is accessed by the query. Henceforth we use the term *error* to refer to the relative

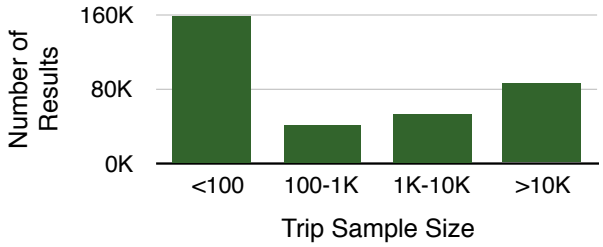


Figure 3.3: Distribution of population sizes for dataset queries.

Figure 3.4: Median error vs population size for queries with no joins (a) and with joins (b).

amount of noise added to query results in order to enforce differential privacy. We use this metric as a proxy for utility since data analysts are primarily concerned with accuracy of results.

Query population size. To evaluate the ability of FLEX to handle both small and large populations, we define a metric called *population size*. The population size of a query is the number of unique trips in the database used to calculate the query results. The population size metric quantifies the extent to which a query targets specific users or trips: a low population size indicates the query is highly targeted, while a higher population size means the query returns statistics over a larger subgroup of records.

Figure 3.3 summarizes the distribution of population sizes of the queries in our dataset. Our dataset contains queries with a wide variety of population sizes, reflecting the diversity of queries in the dataset.

Utility of FLEX-based differential privacy. We evaluate the utility of FLEX by comparing the error introduced by differential privacy on each query against the population size of that query. For small population sizes, we expect our approach to protect privacy by producing high error; for large population sizes, we expect our approach to provide high utility by producing low error.

We used FLEX to produce differentially private results for each query in our dataset. We report separately the results for queries with no joins and those with joins. For each cell in the results, we

Query	Description	# Joins
Q1	Billed, shipped, and returned business	0
Q4	Priority system status and customer satisfaction	1
Q13	Relationship between customers and order size	1
Q16	Suppliers capable of supplying various part types	1
Q21	Suppliers with late shipping times for required parts	3

Table 3.2: Evaluated TPC-H queries.

calculated the relative (percent) error introduced by FLEX, as compared to the true (non-private) results. Then, we calculated the median error of the query by taking the median of the error values of all cells. For this experiment, we set $\epsilon = 0.1$ and $\delta = n^{-\epsilon \ln n}$ (where n is the size of the database), following Dwork and Lei [31].

Figure 3.4 shows the median error of each query against the population size of that query for queries with no joins (a) and with joins (b). The results indicate that FLEX achieves its primary goal of supporting joins. Figure 3.4 shows similar trends with and without joins. In both cases the median error generally decreases with increasing population size; furthermore, the magnitude of the error is comparable for both. Overall, FLEX provides high utility (less than 10% error) for a majority of queries both with and without joins.

Figure 3.4(b) shows a cluster of queries with higher errors but exhibiting the same error-population size correlation as the main group. The queries in this cluster perform many-to-many joins on private tables and do not benefit from the public table optimization described in Section 3.2. Even with this upward shift, a high utility is predicted for sufficiently large population size: at population sizes larger than 5 million the median error drops below 10%.

Hay et al. [43] define the term *scale-ϵ exchangeability* to describe the trend of decreasing error with increasing population size. The practical implication of this property is that a desired utility can always be obtained by using a sufficiently large population size. For counting queries, a local sensitivity-based mechanism using Laplace noise is expected to exhibit scale-ϵ exchangeability. Our results provide empirical confirmation that FLEX preserves this property, for both queries with and without joins, while calculating an approximation of local sensitivity.

Utility of FLEX on TPC-H benchmark

We repeat our utility experiment using TPC-H [25], an industry-standard SQL benchmark. The source code and data for this experiment are available for download [5].

The TPC-H benchmark includes synthetic data and queries simulating a workload for an archetypal industrial company. The data is split across 8 tables (customers, orders, suppliers, etc.) and the benchmark includes 22 SQL queries on these tables.

The TPC-H benchmark is useful for evaluating our system since the queries are specifically chosen to exhibit a high degree of complexity and to model typical business decisions [25]. This experiment measures the ability of our system to handle complex queries and provide high utility in a new domain.

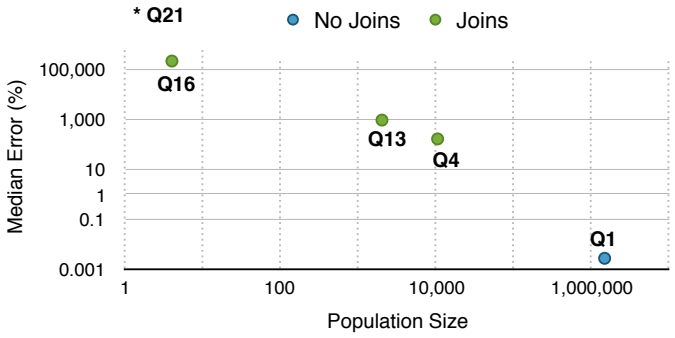


Figure 3.5: Median error vs population size (TPC-H queries). *Query 21 is supported, but due to low coverage and high sensitivity, its relative error is much higher than that of the other queries.

Experiment setup. We populated a database using the TPC-H data generation tool with the default scale factor of 1. We selected the counting queries from the TPC-H query workload, resulting in five queries for evaluation including three queries that use join. The selected queries use SQL's GROUP BY operator and other SQL features including filters, order by, and subqueries. The selected queries are summarized in Table 3.2. The remaining queries in the benchmark are not applicable for this experiment as they return raw data or use non-counting aggregation functions.

We computed the median population size and median error for each query using the same methodology as the previous experiment and privacy parameters $\epsilon = 0.1$ and $\delta = 1/n^2$. We marked as private every table containing customer or supplier information (customer, orders, lineitem, supplier, partsupp). The 3 tables containing non-sensitive metadata (region, nation, part) were marked as public.

Results. The results are presented in Figure 3.5. Elastic sensitivity exhibits the same trend as the previous experiment: error decreases with increasing population size; this trend is observed for queries with and without joins, but error tends to be higher for queries with many joins.

Inherently sensitive queries

Differential privacy is designed to provide good utility for statistics about large populations in the data. Queries with low population size, by definition, pose an inherent privacy risk to individuals; differential privacy *requires* poor utility for their results in order to protect privacy. As pointed out by Dwork and Roth [35], "Questions about specific individuals cannot be safely answered with accuracy, and indeed one might wish to reject them out of hand."

Since queries with low population size are inherently sensitive and therefore not representative of the general class of queries of high interest for differential privacy, we exclude queries with sample size smaller than 100 in the remaining experiments. This ensures the results reflect the behavior of FLEX on queries for which high utility may be expected.

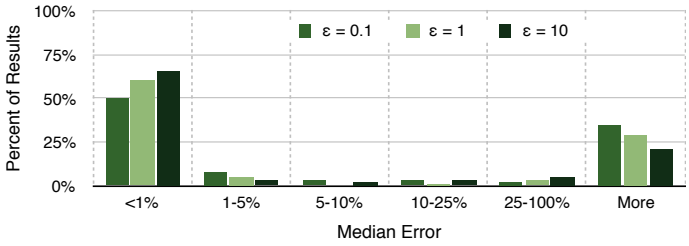


Figure 3.6: Effect of ϵ on median error.

Effect of Privacy Budget

In this section we evaluate the effect of the privacy budget on utility of FLEX-based differential privacy. For each value of ϵ in the set $\{0.1, 1, 10\}$ (keeping δ fixed at $n^{-\epsilon \ln n}$), we computed the median error of each query, as in the previous experiment.

We report the results in Figure 3.6, as a histogram grouping queries by median error. As expected, larger values of ϵ result in lower median error. When $\epsilon = 0.1$, FLEX produces less than 1% median error for approximately half (49.8%) of the less sensitive queries in our dataset.

High-error queries. The previous two experiments demonstrate that FLEX produces good utility for queries with high population size, but as demonstrated by the number of queries in the "More" bin in Figure 3.6, FLEX also produces high error for some queries.

To understand the root causes of this high error, we manually examined a random sample of 50 of these queries and categorized them according to the primary reason for the high error.

We summarize the results in Table 3.3. The category *filter on individual's data* (8% of high error queries) includes queries that use a piece of data specific to an individual—either to filter the sample with a `Where` clause, or as a histogram bin. For example, the query might filter the set of trips by comparing the trip's driver ID against a string literal containing a particular driver's ID, or it might construct a histogram grouped by the driver ID, producing a separate bin for each individual driver. These queries are designed to return information specific to individuals.

The category *low-population statistics* (72% of high error queries) contains queries with a `Where` clause or histogram bin label that shrinks the set of rows considered. A query to determine the success rate of a promotion might restrict the trips considered to those within a small city, during the past week, paid for using a particular type of credit card, and using the promotion. The analyst in this case may not intend to examine the information of any individual, but since the query is highly dependent on a small set of rows, the results may nevertheless reveal an individual's information.

These categories suggest that even queries with a population size larger than 100 can carry inherent privacy risks, therefore differential privacy requires high error for the reasons motivated earlier.

The third category (20% of high error queries) contains queries that have many-to-many joins with large maximum frequency metrics and which do not benefit from any of the optimizations

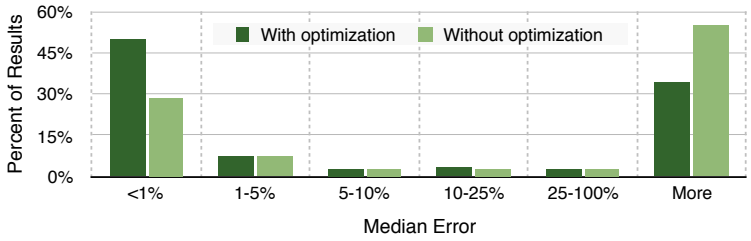


Figure 3.7: Impact of public table optimization.

Category	Percent
Filters on individual's data	8%
Low-population statistics	72%
Many-to-many `Join` causes high elastic sensitivity	20%

Table 3.3: Manual categorization of queries with high error.

described in Section 3.2. These queries are not necessarily inherently sensitive; the high error may be due to a loose bound on local sensitivity arising from elastic sensitivity's design.

Impact of Public Table Optimization

Section 3.2 describes an optimization for queries joining on public tables. We measure the impact of this optimization on query utility by calculating median error introduced by FLEX for each query in our dataset with the optimization enabled and disabled. We use the same experimental setup described in the previous section, with $\epsilon = 0.1$ and $\delta = n^{-\epsilon \ln n}$. As before, we exclude queries with population size less than 100.

The optimization is applied to 23.4% of queries in our dataset. Figure 3.7 shows the utility impact of the optimization across all queries. The optimization increases the percentage of queries with greatest utility (error less than 1.0%) from 28.5% to 49.8%. The majority of the increase in high-utility queries come from the lowest-utility bin (error greater than 100%) while little change is seen in the mid-range error bins. This suggests our optimization is most effective on queries which would otherwise produce high error, optimizing more than half of these queries into the 1% error bin.

Comparison with wPINQ

We aim to compare our approach to alternative differential privacy mechanisms with equivalent support for real-world queries. Of the mechanisms listed in Chapter 2.4, only wPINQ supports counting queries with the full spectrum of join types.

Since wPINQ programs are implemented in C#, we are unable to run wPINQ directly on our SQL query dataset. Instead we compare the utility between the two mechanisms for a selected set of representative queries. The precise behavior of each mechanism may differ for every query,

	Program	Joined tables	Median Population Size	Median Error (%)	
				wPINQ	Elastic Sensitivity
1.	Count distinct drivers who have completed a trip in San Francisco yet enrolled as a driver in a different city.	trips, drivers	663	45.9	22.6
2.	Count driver accounts that are active and were tagged after June 6 as duplicate accounts.	users, user_tags	734	71.5	2.8
3.	Count motorbike drivers in Hanoi who are currently active and have completed 10 or more trips.	drivers, analytics	212	51.4	4.72
4.	Histogram: Daily trips by city (for all cities) on Oct. 24, 2016.	trips, cities	87	11.5	23
5.	Histogram: Total trips per driver in Hong Kong between Sept. 9 and Oct. 3, 2016.	trips, drivers	1	974	6437
6.	Histogram: Drivers by different thresholds of total completed trips for drivers registered in Syndey, AUS who have completed a trip within the past 28 days.	drivers, analytics	72	51.5	27.8

Table 3.4: Utility comparison of wPINQ and FLEX for selected set of representative counting queries using join.

however this experiment provides a relative comparison of the mechanisms for the most common cases.

Experiment Setup. We selected a set of representative queries based on the most common equijoin patterns (joined tables and join condition) across all counting queries in our dataset. We identify the three most common join patterns for both histogram and non-histogram queries and select a random query representing each. Our six selected queries collectively represent 8.6% of all join patterns in our dataset.

For each selected query we manually transcribe the query into a wPINQ program. To ensure a fair comparison, we use wPINQ's `select` operator rather than the `join` operator for joins on a public table. This ensures that no noise is added to protect records in public tables, equivalent to the optimization described in Section 3.2.

Our input data for these programs includes all records from the cities table, which is public, and a random sample of 1.5 million records from each private table (it was not feasible to download the full tables, which contain over 2 billion records). We execute each program 100 times with the wPINQ runtime [11].

To obtain baseline (non-differentially private) results we run each SQL query on a database populated with only the sampled records. For elastic sensitivity we use max-frequency metrics calculated from this sampled data. We compute the median error for each query using the methodology described in the previous section, setting $\epsilon = 0.1$ for both mechanisms.

Table 3.4 summarizes the queries and median error results. FLEX provides lower median error than wPINQ for programs 1, 2, 3 and 6—more than 90% lower for 2 and 3 and nearly 50% lower for programs 1 and 6. FLEX produces higher error than wPINQ for programs 4 and 5.

In the case of program 5, both mechanisms produce errors above 900%. The median population size of 1 for this program indicates that our experiment data includes very few trips *per driver*

that satisfy the filter conditions. Elastic sensitivity provides looser bounds on local sensitivity for queries that filter more records, resulting in a comparably higher error for queries such as this one. Given that such queries are inherently sensitive, a high error (low utility) is required for *any* differential privacy mechanism, therefore the comparably higher error of FLEX is likely insignificant in practice.

Proserpio et al. [63] describe a post-processing step for generating synthetic data by using wPINQ results to guide a Markov-Chain Monte Carlo simulation. The authors show that this step improves utility for graph triangle counting queries when the original query is executed on the synthetic dataset. While this strategy may produce higher utility than the results presented in Table 3.4, we do not evaluate wPINQ with this additional step since it is not currently automated.

3.5 Related Work

Differential privacy was originally proposed by Dwork [29, 30, 32], and the reference by Dwork and Roth [35] provides an excellent general overview of differential privacy. Much of this work focuses on mechanisms for releasing the results of specific algorithms. Our focus, in contrast, is on a general-purpose mechanism for SQL queries that supports general equijoins. We survey the existing general mechanisms that support join in Chapter 2.

Lu et al. [53] propose a mechanism for generating differentially private synthetic data such that queries with joins have similar *performance characteristics*, but not necessarily similar answers, on the synthetic and true databases. However, Lu et al. do not propose a mechanism for answering queries with differential privacy. As such, it does not satisfy either of the two requirements in Section 2.3.

Airavat [65] enforces differential privacy for arbitrary MapReduce programs, but requires the analyst to bound the range of possible outputs of the program, and clamps output values to lie within that range. Fuzz [38, 40] enforces differential privacy for functional programs, but does not support one-to-many or many-to-many joins.

Propose-test-release [31] (PTR) is a framework for leveraging local sensitivity that works for arbitrary real-valued functions. PTR requires (but does not define) a way to calculate the local sensitivity of a function. Our work on elastic sensitivity is complementary and can enable the use of PTR by providing a bound on local sensitivity.

Sample & aggregate [59, 60] is a data-dependent framework that applies to all statistical estimators. It works by splitting the database into chunks, running the query on each chunk, and aggregating the results using a differentially private algorithm. Sample & aggregate cannot support joins, since splitting the database breaks join semantics, nor does it support queries that are not statistical estimators, such as counting queries. GUPT [56] is a practical system that leverages the sample & aggregate framework to enforce differential privacy for general-purpose analytics.

The *Exponential Mechanism* [54] supports queries that produce categorical (rather than numeric) data. It works by randomly selecting from the possible outputs according to a *scoring function* provided by the analyst. Extending FLEX to support the exponential mechanism would require specification of the scoring function and a means to bound its sensitivity.

A number of less-general mechanisms for performing specific graph analysis tasks have been proposed [42, 46, 47, 66]. These tasks often involve joins, but the mechanisms used to handle them are specific to the task and are not applicable for general-purpose analytics. For example, the recursive mechanism [23] supports general equijoins in the context of graph analyses, but is restricted to monotonic queries and in the worst case, runs in time exponential in the number of participants in the database.

Kifer et al. [48] point out that database constraints (such as uniqueness of a primary key) can lead to leaks of private data. Such constraints are common in practice, and raise concerns for *all* differential privacy approaches. Kifer et al. propose increasing sensitivity based on the specific constraints involved, but calculating this sensitivity is computationally hard. Developing a tractable method to account for common constraints, such as primary key uniqueness, is an interesting target for future work.

Chapter 4

Differential Privacy via Query Rewriting

4.1 Introduction

The previous chapter described elastic sensitivity, a differential privacy mechanism that supports SQL queries with joins. Elastic sensitivity significantly expands the scope of differential privacy for SQL but it alone does not provide practical differential privacy across the full spectrum of queries surveyed in Chapter 2. For example, elastic sensitivity works well for counting queries but not as well for other aggregation functions.

This is emblematic of a general trend in mechanism design. As summarized in Table 2.1, each mechanism supports a specific class of queries. Furthermore, the best mechanism for any particular query often depends on the dataset [43]. Consequently, a practical solution for differential privacy must provide flexibility for mechanism selection, as there is not an optimal mechanism that can provide practical differential privacy across all queries and datasets. Such a solution would then satisfy Requirement 3 by supporting all aggregation functions via selection of an appropriate mechanism.

This chapter describes CHORUS, a system with a novel architecture for providing differential privacy for statistical SQL queries. CHORUS supports multiple mechanisms simultaneously while integrating easily into existing environments, meeting all three requirements established in Chapter 2.

The key insight of our approach is to embed the differential privacy mechanism into the SQL query before execution, so the query automatically enforces differential privacy on its own output. We define a SQL query with this property as an *intrinsically private query*. CHORUS automatically converts untrusted input queries into intrinsically private queries. This approach enables a new architecture in which queries are rewritten, then submitted to an unmodified database management system (DBMS).

This new architecture addresses the major challenges outlined in Chapter 1. First, CHORUS is compatible with any SQL database that supports standard math functions ($rand$, ln, etc.) thus avoiding the need for a custom runtime or modifications to the database engine. By using a standard SQL database engine instead of a custom runtime, CHORUS can leverage the reliability, scalability

and performance of modern databases, which are built on decades of research and engineering experience.

Second, CHORUS enables the modular implementation and adoption of many different mechanisms, supporting a significantly higher percentage of queries than any single mechanism and providing increased flexibility for both general and specialized use cases. CHORUS automatically selects a mechanism for each input query based on an extensible set of selection rules. To the best of our knowledge, no existing system provides these capabilities.

CHORUS also protects against an untrusted analyst: even if the submitted query is malicious, our transformation rules ensure that the executed query returns only differentially private results. The results can therefore be returned directly to the analyst without post-processing. This enables easy integration into existing data environments via a single pre-processing step.

We demonstrate the CHORUS approach with four existing general-purpose differential privacy mechanisms: Elastic Sensitivity [45], Restricted Sensitivity [20], Weighted PINQ [32] and Sample & Aggregate [56]. These mechanisms support a range of SQL features and analytic tasks. CHORUS contains query transformation rules for each mechanism which convert untrusted (non–intrinsically private) queries into intrinsically private queries. We also describe how additional mechanisms can be added to CHORUS.

Deployment. CHORUS is currently being deployed at Uber for its internal analytics tasks. CHORUS represents a significant part of the company's General Data Protection Regulation (GDPR) [6] compliance, and provides both differential privacy and access control enforcement. We have made CHORUS available as open source [3].

Evaluation. In the first evaluation of its kind, we use CHORUS to evaluate the utility and performance of the selected mechanisms on real data. Our dataset includes 18,774 real queries written by analysts at Uber.

Contributions

In summary, we make the following contributions:

1. We present CHORUS, representing a novel architecture for enforcing differential privacy on SQL queries that simultaneously supports a wide variety of mechanisms and runs on any standard SQL database (Section 4.3).

2. We describe and formalize the novel use of rule-based query rewriting to automatically transform an untrusted SQL query into an intrinsically private query using four example general-purpose mechanisms. We describe how other mechanisms can be supported using the same approach (Section 4.4, 4.5).

3. We report on our experience deploying CHORUS to enforce differential privacy at Uber, where it processes more than 10,000 queries per day (Section 4.6).

Mechanism	General-Purpose	Strengths	Supported Constructs	Sensitivity Measure	Algorithmic Requirements
Elastic Sensitivity [45]	✓	General analytics	Counting queries w/ joins	Local	Laplace noise
Restricted Sensitivity [20]	✓	Graph analysis	Counting queries w/ joins	Global	Laplace noise
WPINQ [63]	✓	Synthetic data gen.	Counting queries w/ joins	Global	Weighted dataset operations, Laplace noise
Sample & Aggregate [68]	✓	Statistical estimators	Single-table aggregations	Local	Subsampling, Widened Winsorized mean, Laplace noise

Table 4.1: Differential privacy mechanisms

4. We use CHORUS to conduct the first large-scale empirical evaluation of the utility and performance of multiple general-purpose differential privacy on real queries and data (Section 4.7).

4.2 Background

Differential privacy provides a formal guarantee of *indistinguishability*. This guarantee is defined in terms of a *privacy budget* ϵ—the smaller the budget, the stronger the guarantee. The formal definition of differential privacy is written in terms of the *distance* $d(x, y)$ between two databases, i.e. the number of entries on which they differ: $d(x, y) = |\{i : x_i \neq y_i\}|$. Two databases x and y are *neighbors* if $d(x, y) = 1$. A randomized mechanism $\mathcal{K} : D^n \to \mathbb{R}$ preserves (ϵ, δ)-differential privacy if for any pair of neighboring databases $x, y \in D^n$ and set S of possible outputs:

$$\Pr[\mathcal{K}(x) \in S] \leq e^{\epsilon}\Pr[\mathcal{K}(y) \in S] + \delta$$

Differential privacy can be enforced by adding noise to the non-private results of a query. The scale of this noise depends on the *sensitivity* of the query. The literature considers two different measures of sensitivity: *global* [32] and *local* [59], as summarized in Section 3.2.

Statistical queries. Differential privacy aims to protect the privacy of individuals in the context of *statistical queries*. In SQL terms, these are queries using standard aggregation operators (`COUNT`, `AVG`, etc.) as well as histograms created via the `GROUP BY` operator in which aggregations are applied to records within each group. Differential privacy is not suitable for queries that return raw data (e.g. rows in the database) since such queries are inherently privacy-violating. We formalize the targeted class of queries in Section 4.5 and discuss how our approach supports histogram queries, which require special care to avoid leaking information via the presence of absence of groups.

Mechanism design. Research on differential privacy has produced a large and growing number of differential privacy mechanisms. Some mechanisms are designed to provide broad

support for many types of queries [20, 32, 54–56, 59, 63], while others are designed to produce maximal utility for a particular application [18, 23, 24, 37, 41–43, 46, 47, 50, 52, 64, 66, 71–73].

While mechanisms adopt unique strategies for enforcing differential privacy in their target domain, they generally share a common set of design choices and algorithmic components. For example, many mechanisms require addition of Laplace noise to the result of the query.

Our approach is motivated by the observation that a wide range of distinct mechanisms can supported with a common set of algorithmic building blocks. In this chapter we formalize several example building blocks via *transformation rules* that describe how each algorithmic component can be embedded within a SQL query. We demonstrate the flexibility of this design by showing that each mechanism can be implemented simply by composing these transformation rules according to the mechanism's definition.

General-purpose mechanisms. The four mechanisms in Table 4.1 are general-purpose because they support a broad range of queries, including commonly used SQL constructs such as join. This chapter focuses on these four mechanisms. Many more specialized mechanisms have substantially similar algorithmic requirements and can be supported as intrinsically private queries using variations of the transformation rules introduced in this chapter. Section 4.7 discusses this subject in more detail.

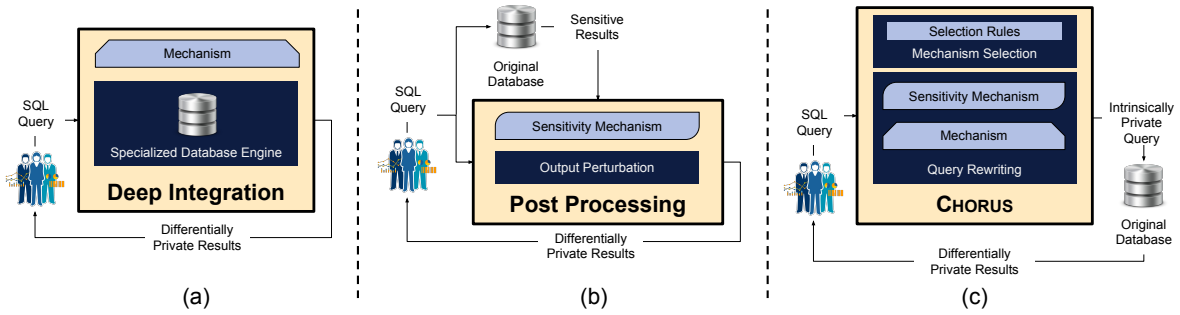
4.3 The CHORUS Architecture

This section presents the system architecture of CHORUS and compares it against existing architectures for differentially private analytics. We first identify the design goals motivating the CHORUS architecture then describe the limitations of existing architectures preventing previous work from attaining these goals. Finally, we describe the novel architecture of CHORUS and provide an overview of our approach.

Existing Architectures

Existing systems for enforcing differential privacy for data analytics tasks adopt one of two architecture types: they are either *deeply integrated* systems or *post processing* systems. These architectures are summarized in Figure 4.1(a) and Figure 4.1(b). PINQ [55], Weighted PINQ [63], GUPT [56], and Airavat [65] follow the *deep integration* architecture: each one provides its own specialized DBMS, and cannot be used with a standard DBMS.

Elastic sensitivity [45] uses the *post processing* architecture, in which the original query is run on the database and noise is added to the final result. This approach supports mechanisms that do not modify the semantics of the original query (PINQ and Restricted sensitivity [20] could also be implemented this way), and has the advantage that it is compatibile with existing DBMSs. However, the post processing architecture is *fundamentally incompatible* with mechanisms that change how the original query executes—including WPINQ and Sample & Aggregate, listed in Table 4.1.



Pros:

- Broad mechanism support

Cons:

- Higher software complexity
- New system required for each mechanism

Pros:

- DBMS-independent

Cons:

- Supports only post-processing mechanisms

Pros:

- DBMS-independent
- Broad mechanism support

Figure 4.1: Existing architectures: (a), (b); architecture of CHORUS: (c).

The *deeply integrated* and *post processing* architectures in Figure 4.1(a) and (b) therefore both fail to address two major challenges in implementing a practical system for differentially private data analytics:

- Custom DBMSs are unlikely to achieve parity with mature DBMSs for a wide range of features including rich SQL support, broad query optimization, high-performance transaction support, recoverability, scalability and distribution.
- Neither architecture supports the simultaneous application of a large number of different mechanisms. The deeply integrated architecture requires building a new DBMS for each mechanism, while the post processing architecture is inherently incompatible with some mechanisms.

The CHORUS Architecture

In CHORUS, we propose a novel alternative to the deeply integrated and post processing architectures used by existing systems. As shown in Figure 4.1(c), CHORUS transforms the input query into an *intrinsically private query*, which is a standard SQL query whose results are *guaranteed to be differentially private*.

An intrinsically private query provides this guarantee by embedding a differential privacy mechanism in the query itself. When executed on an unmodified SQL database, the embedded privacy mechanism ensures that the query's results preserve differential privacy. The approach has three key advantages over previous work:

- CHORUS is DBMS-independent (unlike the deeply integrated approach): it requires neither modifying the database nor switching to purpose-built database engines. Our approach can therefore leverage existing high-performance DBMSs to scale to big data.
- CHORUS can implement a wide variety of privacy-preserving techniques. Unlike the post processing approach, CHORUS is compatible with all of the mechanisms listed in Table 4.1, and many more.
- CHORUS eliminates the need for post-processing, allowing easy integration in existing data processing pipelines. Our approach enables a single data processing pipeline for all mechanisms.

CHORUS's architecture is specifically designed to be easily integrated into existing data environments. We report on the deployment of CHORUS at Uber in Section 4.6.

Constructing intrinsically private queries. The primary challenge in implementing this architecture is transforming untrusted queries into intrinsically private queries. This process must be flexible enough to support a wide variety of privacy mechanisms and also general enough to support ad-hoc SQL queries.

Constructing intrinsically private queries automatically has additional advantages. This approach protects against malicious analysts by guaranteeing differentially private results by construction. It is also transparent to the analyst since it does not require input from the analyst to preserve privacy or select a privacy mechanism.

As shown in Figure 4.1(c), CHORUS constructs intrinsically private queries in two steps:

1. The *Mechanism Selection* component automatically selects an appropriate differential privacy mechanism to apply.
2. The *Query Rewriting* component embeds the selected mechanism in the input query, transforming it into an intrinsically private query.

To select a mechanism, CHORUS leverages an extensible mechanism selection engine, discussed next. The query rewriting step then transforms the input query to produce an intrinsically private query embedding the selected mechanism. For this step, we employ a novel use of rule-based query rewriting, which has been studied extensively for query optimization but, to our knowledge, never applied to differential privacy. We introduce our solution by example in Section 4.4 and formalize it in Section 4.5. This dissertation focuses on differential privacy, but the same approach could be used to enforce other types of privacy guarantees or security policies [69].

Mechanism selection. CHORUS implements an extensible *mechanism selection engine* that automatically selects a differential privacy mechanism for each input query. This engine can be extended based on available mechanisms, performance and utility goals, and to support custom

mechanism selection techniques. For example, Hay et al. [49] demonstrate that a machine learning-based approach can leverage properties of the data to select a mechanism most likely to yield high utility. CHORUS is designed to support any such approach. We describe an example of a syntax-based mechanism selection in Section 4.6.

Privacy budget management. CHORUS does not prescribe a specific privacy budget management strategy, as the best way to manage the available privacy budget in practice will depend on the deployment scenario and threat model. CHORUS provides flexibility in how the budget is managed: the sole requirement is that rewriters are supplied with the ϵ value apportioned to each query.[1]

For the case of a single global budget, where CHORUS is deployed as the sole interface to the database, CHORUS can track the remaining budget according to the standard composition theorem for differential privacy [32]. When a new query is submitted, CHORUS subtracts from the remaining budget the ϵ value allocated to that query, and refuses to process new queries when the budget is exhausted. In Section 4.7, we discuss more sophisticated methods that may yield better results for typical deployments.

4.4 Query Rewriting

This section demonstrates our approach by example, using the four general-purpose differential privacy mechanisms listed in Table 4.1. For each mechanism, we briefly review the algorithm used. We then show, using simple example queries, how an input SQL query can be systematically transformed into an intrinsically private query embedding each algorithm, and give an argument for the correctness of each transformation.

Sensitivity-Based Mechanisms

We first consider two mechanisms that add noise to the final result of the query: Elastic Sensitivity [45] and Restricted Sensitivity [20]. Elastic Sensitivity is a bound on the local sensitivity of a query, while Restricted Sensitivity is based on global sensitivity. Both approaches add Laplace noise to the query's result.

For a query with sensitivity s returning value v, the Laplace mechanism releases $v + \mathrm{Lap}(s/\epsilon)$, where ϵ is the privacy budget allocated to the query. Given a random variable U sampled from the uniform distribution, we can compute the value of a random variable $X \sim \mathrm{Lap}(s/\epsilon)$:

$$X = -\frac{s}{\epsilon}\,\mathrm{sign}(U)\,\ln(1-2|U|)$$

[1] For simplicity we consider approaches where CHORUS stores the budget directly. Our query rewriting approach could allow the DBMS to assist with budget accounting, for example by storing ϵ values in a separate table and referencing and updating the values within the rewritten query.

In SQL, we can sample from the uniform distribution using `RANDOM()`. Consider the following query, which returns a (non–differentially private) count of trips in the database. This query can be transformed into an intrinsically private query as follows:

```
SELECT COUNT(*) AS count FROM trips
                ⇓
WITH orig AS (SELECT COUNT(*) AS count FROM trips),
     uniform AS (SELECT *, RANDOM()-0.5 AS u
                 FROM orig)
                ⇓
WITH orig AS (SELECT COUNT(*) AS count FROM trips),
     uniform AS (SELECT *, RANDOM()-0.5 AS u
                 FROM orig)
SELECT count-( s/ϵ )*SIGN(u)*LN(1-2*ABS(u)) AS count
FROM uniform
```

The first step defines U using `RANDOM()`, and the second uses U to compute the corresponding Laplace noise.

The correctness of this approach follows from the definition of the Laplace mechanism. The two transformation steps generate Laplace noise with the correct scale, and add it to the sensitive query results.

CHORUS can calculate Elastic Sensitivity with smoothing via smooth sensitivity [59][2] or Restricted Sensitivity via a dataflow analysis of the query. Such an analysis is described in Chapter 3.

We formalize the construction of intrinsically private queries via sensitivity-based approaches using the *Laplace Noise transformation*, defined in Section 4.5.

Weighted PINQ

Weighted PINQ (WPINQ) enforces differential privacy for counting queries with equijoins. A key distinction of this mechanism is that it produces a differentially private *metric* (called a *weight*), rather than a count. These weights are suitable for use in a workflow that generates differentially private synthetic data, from which counts are easily derived. The workflow described in [63] uses weights as input to a Markov chain Monte Carlo (MCMC) process.

CHORUS's implementation of WPINQ computes noisy weights for a given counting query according to the mechanism's definition [63]. Since the weights are differentially private, they can be released to the analyst for use with any desired workflow.

The WPINQ mechanism adds a weight to each row of the database, updates the weights as the query executes to ensure that the query has a sensitivity of 1, and uses the Laplace mechanism to add noise to the weighted query result. WPINQ has been implemented as a standalone data

[2]Smooth sensitivity guarantees ϵ, δ-differential privacy, and incorporates the setting of δ into the smoothed sensitivity value.

processing engine with a specialized query language, but has not been integrated into any SQL DBMS.

Where a standard database is a collection of tuples in D^n, a weighted database (as defined in Proserpio et al. [63]) is a function from a tuple to its weight ($D^n \to \mathbb{R}$). In this setting, counting the number of tuples with a particular property is analogous to summing the weights of all such tuples. Counting queries can therefore be performed using *sum*.

In fact, summing weights in a weighted dataset produces exactly the same result as the corresponding counting query on the original dataset, when the query does not contain joins. When the query does contain joins, WPINQ scales the weight of each row of the join's output to maintain a sensitivity of 1. Proserpio et al. [63] define the weight of each row in a join as follows:

$$A \bowtie B = \sum_k \frac{A_k \times B_k^T}{||A_k|| + ||B_k||} \tag{4.1}$$

Since the scaled weights ensure a sensitivity of 1, Laplace noise scaled to $1/\epsilon$ is sufficient to enforce differential privacy. WPINQ adds noise with this scale to the results of the weighted query.

In SQL, we can accomplish the first task (adding weights) by adding a column to each relation. For example, using our previous example query:

```
SELECT COUNT(*) FROM trips
```

$\Downarrow$

```
SELECT SUM(weight)
FROM (SELECT *, 1 AS weight FROM trips)
```

This transformation adds a weight of 1 to each row in the table, and changes the `COUNT` aggregation function into a `SUM` of the rows' weights. The correctness of this transformation is easy to see: as required by WPINQ [63], the transformed query adds a weight to each row, and uses `SUM` in place of `COUNT`.

We can accomplish the second task (scaling weights for joins) by first calculating the norms $||A_K||$ and $||B_k||$ for each key k, then the new weights for each row using $A_k \times B_k^T$. For a join between the `trips` and `drivers` tables, for example, we can compute the norms for each key:

```
WITH tnorms AS (SELECT driver_id,
                       SUM(weight) AS norm
                FROM trips
                GROUP BY driver_id),
     dnorms AS (SELECT id, SUM(weight) AS norm
                FROM drivers
                GROUP BY id)
```

Then, we join the norms relations with the original results and scale the weight for each row:

```
SELECT ...,
  (t.weight*d.weight)/(tn.norm+dn.norm) AS weight
FROM trips t, drivers d, tnorm tn, dnorm dn
```

```
WHERE t.driver_id = d.id
  AND t.driver_id = tn.driver_id
  AND d.id = dn.id
```

The correctness of this transformation follows from equation (4.1). The relation `tnorms` corresponds to $||A_k||$, and `dnorms` to $||B_k||$. For each key, `t.weight` corresponds to A_k, and `d.weight` to B_k.

Finally, we can accomplish the third task (adding Laplace noise scaled to $1/\epsilon$) as described earlier.

We formalize the construction of intrinsically private queries via WPINQ using three transformations: the *Metadata Propagation* transformation to add weights to each row, the *Replace Aggregation Function* transformation to replace counts with sums of weights, and the *Laplace Noise* transformation to add noise to the results. All three are defined in Section 4.5.

Sample & Aggregate

The Sample & Aggregate [59,68] mechanism works for all statistical estimators, but does not support joins. Sample & Aggregate has been implemented in GUPT [56], a standalone data processing engine that operates on Python programs, but has never been integrated into a practical database. As defined by Smith [68], the mechanism has three steps:

1. Split the database into disjoint *subsamples*
2. Run the query on each subsample independently
3. Aggregate the results using a differentially-private algorithm

For differentially-private aggregation, Smith [68] suggests *Widened Winsorized mean*. Intuitively, Widened Winsorized mean first calculates the *interquartile range*—the difference between the 25th and 75th percentile of the subsampled results. Next, the algorithm *widens* this range to include slightly more data points, then clamps the subsampled results to lie within the widened range. This step eliminates outliers, which is important for enforcing differential privacy. Finally, the algorithm takes the average of the clamped results, and adds Laplace noise scaled to the size of the range (i.e. the effective sensitivity) divided by ϵ.

In SQL, we can accomplish tasks 1 and 2 by adding a `GROUP BY` clause to the original query. Consider a query that computes the average of trip lengths:

```
SELECT AVG(trip_distance) FROM trips
```

$\Downarrow$

```
SELECT AVG(trip_distance), ROW_NUM() MOD  n  AS samp
FROM trips
GROUP BY samp
```

This transformation generates n subsamples and runs the original query on each one. The correctness of tasks 1 and 2 follows from the definition of subsampling: the `GROUP BY` ensures that the subsamples are disjoint, and that the query runs on each subsample independently. To accomplish task 3 (differentially private aggregation), we can use a straightforward implementation of Widened Winsorized mean in SQL, since the algorithm itself is the same for each original query.

We formalize the construction of intrinsically private queries via Sample & Aggregate using the *Subsampling* transformation, defined in the next section.

4.5 Formalization & Correctness

This section formalizes the construction of intrinsically private queries as introduced in Section 4.4. We begin by introducing notation, then define reusable transformation rules that can be composed to construct mechanisms. Next, we formalize the four mechanisms described earlier in terms of these rules. Finally, we prove a correctness property: our transformations do not modify the semantics of the input query.

By formalizing the transformation rules separately from the individual mechanisms, we allow the rules to be re-used when defining new mechanisms, taking advantage of the common algorithmic requirements demonstrated in Table 4.1. An additional benefit of this approach is the ability to prove correctness properties of the rules themselves, so that these properties extend to all mechanisms implemented using the rules.

Preliminaries

Core relational algebra. We formalize our transformations as rewriting rules on a core relational algebra that represents general statistical queries. We define the core relational algebra in Figure 4.2. This algebra includes the most commonly-used features of query languages like SQL: selection (σ), projection (Π), equijoins ($\bowtie$), and counting with and without aggregation. We also include several features specifically necessary for our implementations: constant values, random numbers, and the arithmetic functions *ln*, *abs*, and *sign*.

We use standard notation for relational algebra with a few exceptions. We extend the projection operator Π to attribute expressions, which allows projection to add new named columns to a relation. For example, if relation r has schema U, then the expression $\Pi_{U \cup \text{weight}:1} r$ adds a new column named "weight" with the default value 1 for each row to the existing columns in r. In addition, we combine aggregation and grouping, writing $\underset{a_1..a_n}{\textit{Count}}$ to indicate a counting aggregation with grouping by columns $a_1..a_n$. We write $\underset{a_1..a_n}{\textit{Sum}_a}$ to indicate summation of column a, grouping by columns $a_1..a_n$.

Notation for rewriting rules. Each transformation is defined as a set of inference rules that rewrites a relational algebra expression. A particular rule allows rewriting an expression as specified in its conclusion (below the line) if the conditions specified in its antecedent (above the

Attribute expressions

$$
\begin{array}{lcl}
a & & \textit{attribute name} \\
e & ::= & a \mid a : v \\
v & ::= & a \mid n \in \mathbb{N} \mid v_1 + v_2 \mid v_1 * v_2 \mid v_1 / v_2 \\
 & \mid & \textit{rand}() \mid \textit{randInt}(n) \mid \textit{ln}(v) \mid \textit{abs}(v) \mid \textit{sign}(v)
\end{array}
$$

Relational transformations

$$
\begin{array}{lcl}
R & ::= & t \mid R_1 \underset{x=y}{\bowtie} R_2 \mid \Pi_{e_1,\ldots,e_n} R \mid \sigma_{\varphi} R \\
 & \mid & \textit{Count}(R) \mid \underset{a_1..a_n}{\textit{Count}}(R)
\end{array}
$$

Selection predicates

$$
\begin{array}{lcl}
\varphi & ::= & e_1 \theta e_2 \mid e \theta v \\
\theta & ::= & < \mid \leq \mid = \mid \neq \mid \geq \mid >
\end{array}
$$

Queries

$$
Q ::= \textit{Count}(R) \mid \underset{a_1..a_n}{\textit{Count}}(R) \mid \textit{Sum}_a(R) \mid \underset{a_1..a_n}{\textit{Sum}_a}(R)
$$

Figure 4.2: Syntax of core relational algebra

line) are satisfied (either through syntactic properties of the query or by applying another inference rule).

Our approach relies on the ability to analyze and rewrite SQL queries. This rewriting can be achieved by a rule-based query optimization engine [1, 7].

Most of the conditions specified in our rewriting rules use standard notation. One exception is conditions of the form $Q : U$, which we use to denote that the query Q results in a relation with schema U. We extend this notation to denote the schemas of database tables (e.g. $t : U$) and relational expressions (e.g. $r_1 : U$).

Some of our rewriting rules have global parameters, which we denote by setting them above the arrow indicating the rewriting rule itself. For example, $r \xrightarrow{x} \Pi_x r$ allows rewriting r to project only the column named x, where the value of x is provided as a parameter. Most parameters are values, but parameters can also be functions mapping a relational algebra expression to a new expression. For example, $r \xrightarrow{f} f(r)$ indicates rewriting r to the result of $f(r)$.

Transformation Rules

Laplace Noise. All four mechanisms in Table 4.1 require generating noise according to the Laplace distribution. We accomplish this task using the *Laplace Noise* transformation, defined in Figure 4.3. This transformation has one parameter: γ, which defines the scale of the noise to be added to the query's result. For a query with sensitivity 1 and privacy budget ϵ, for example, $\gamma = 1/\epsilon$ suffices to enforce differential privacy.

The Laplace Noise transformation defines a single inference rule. This rule allows rewriting a top-level query with schema U according to the Lap function. Lap wraps the query in two

$$\boxed{Q \xrightarrow{\gamma} Q} \qquad \frac{Q : U}{Q \xrightarrow{\gamma} \mathrm{Lap}(Q)}$$

where

$$\mathrm{Unif}(Q) = \Pi_{U \cup \{\mathrm{u}_x : \mathit{rand}()-0.5 \mid x \in U\}}(Q))$$
$$\mathrm{Lap}(Q) = \Pi_{\{x : x + - \gamma\, \mathit{sign}(\mathrm{u}_x)\mathit{ln}(1-2\,\mathit{abs}(\mathrm{u}_x)) \mid x \in U\}}(\mathrm{Unif}(Q))$$

Figure 4.3: Laplace Noise Transformation

projection operations; the first (defined in Unif) samples the uniform distribution for each value in the result, and the second (defined in Lap) uses this value to add noise drawn from the Laplace distribution.

Metadata Propagation. Many mechanisms require tracking metadata about each row as the query executes. To accomplish this, we define the *Metadata Propagation* transformation in Figure 4.4. The Metadata Propagation transformation adds a column to each referenced table and initializes its value for each row in that table, then uses the specified composition functions to compose the values of the metadata column for each resulting row of a join or an aggregation.

The Metadata Propagation transformation has three parameters: i, a function defining the initial value of the metadata attribute for each row in a database table; j, a function specifying how to update the metadata column for a join of two relations; and c, a function specifying how to update the metadata column for subqueries.

The inference rule for a table t uses projection to add a new attribute to t's schema to hold the metadata, and initializes that attribute to the value of $i()$. The rules for projection and selection simply propagate the new attribute. The rule for joins applies the j function to perform a localized update of the metadata column. The rules for counting subqueries invoke the update function c to determine the new value for the metadata attribute. Finally, the rules for counting queries eliminate the metadata attribute to preserve the top-level schema of the query.

Replacing Aggregation Functions. The *Replace Aggregation Function* transformation, defined in Figure 4.5, allows replacing one aggregation function (Γ) with another (Γ'). To produce syntactically valid output, Γ and Γ' must be drawn from the set of available aggregation functions. The antecedent is empty in the rewriting rules for this transformation because the rules operate only on the outermost operation of the query.

Subsampling. The *Subsampling* transformation, defined in Figure 4.6, splits the database into disjoint subsamples, runs the original query on each subsample, and aggregates the results according to a provided function. The Subsampling transformation is defined in terms of the Metadata Propagation transformation, and can be used to implement partition-based differential privacy mechanisms like Sample & Aggregate.

The parameters for the Subsampling transformation are $\mathcal{A}$, a function that aggregates the subsampled results, and n, the number of disjoint subsamples to use during subsampling. Both in-

$$\boxed{R \overset{i,j,c}{\to} R} \qquad \frac{r_1 \overset{i,j,c}{\to} r_1' \quad r_2 \overset{i,j,c}{\to} r_2' \quad r_1' : U_1 \quad r_2' : U_2}{r_1 \underset{A=B}{\bowtie} r_2 \overset{i,j,c}{\to} j(r_1' \underset{A=B}{\bowtie} r_2')}$$

$$\frac{t : U \quad \mathrm{m} \notin U}{t \overset{i,j,c}{\to} \Pi_{U \cup \mathrm{m}:i()} t} \qquad \frac{r \overset{i,j,c}{\to} r'}{\Pi_U(r) \overset{i,j,c}{\to} \Pi_{U \cup \{\mathrm{m}\}}(r')}$$

$$\frac{r \overset{i,j,c}{\to} r'}{\sigma_\phi(r) \overset{i,j,c}{\to} \sigma_\phi(r')} \qquad \frac{r \overset{i,j,c}{\to} r' \quad \mathit{Count}(r') : U}{\mathit{Count}(r) \overset{i,j,c}{\to} \Pi_{U \cup \mathrm{m}:c(r')} \mathit{Count}(r')}$$

$$\frac{r \overset{i,j,c}{\to} r' \quad \underset{a_1..a_n}{\mathit{Count}}(r') : U}{\underset{a_1..a_n}{\mathit{Count}}(r) \overset{i,j,c}{\to} \Pi_{U \cup \mathrm{m}:c(r')} \underset{a_1..a_n}{\mathit{Count}}(r')}$$

$$\boxed{Q \overset{i,j,c}{\to} Q} \qquad \frac{r \overset{i,j,c}{\to} r'}{\mathit{Count}(r) \overset{i,j,c}{\to} \mathit{Count}(r')} \qquad \frac{r \overset{i,j,c}{\to} r'}{\underset{a_1..a_n}{\mathit{Count}}(r) \overset{i,j,c}{\to} \underset{a_1..a_n}{\mathit{Count}}(r')}$$

Figure 4.4: Metadata Propagation Transformation

$$\boxed{Q \overset{\Gamma,\Gamma'}{\to} Q} \qquad \frac{}{\Gamma(r) \overset{\Gamma,\Gamma'}{\to} \Gamma'(r)} \qquad \frac{}{\underset{a_1..a_n}{\Gamma}(r) \overset{\Gamma,\Gamma'}{\to} \underset{a_1..a_n}{\Gamma'}(r)}$$

Figure 4.5: Replace Aggregation Function Transformation

ference rules defined by the transformation rewrite the queried relation using the Metadata Propagation transformation ($r \overset{i,j,c}{\to} r'$). The parameters for Metadata Propagation initialize the metadata attribute with a random *subsample number* between zero and n, and propagate the subsample number over counting subqueries. The update functions for joins and counting subqueries is undefined, because subsampling is incompatible with queries containing these features.[3]

In order to satisfy the semantics preservation property, the aggregation function $\mathcal{A}$ must transform the query results on each subsample into a final result with the same shape as the original query. Many aggregation functions satisfy this property (e.g. the mean of all subsamples), but not all of them provide differential privacy.

Mechanism Definitions

We now formally define the mechanisms described earlier in terms of the transformations defined earlier. We describe each mechanism as a series of one or more transformations, and define the

[3]Subsampling changes the semantics of joins, since join keys in separate partitions will be prevented from matching. It also changes the semantics of aggregations in subqueries, since the aggregation is computed within a single subsample.

$$\boxed{Q \xrightarrow{\mathcal{A}} Q} \qquad \frac{r \overset{i,j,c}{\rightarrow} r'}{\mathit{Count}(r) \xrightarrow{\mathcal{A}} \mathcal{A}(\underset{m}{\mathit{Count}}(r'))}$$

$$\frac{r \overset{i,j,c}{\rightarrow} r'}{\underset{G_1..G_n}{\mathit{Count}}(r) \xrightarrow{\mathcal{A}} \mathcal{A}(\underset{G_1..G_n,m}{\mathit{Count}}(r'))}$$

where $i = \mathit{randInt}(n), j = \perp, c(r) = \perp$, and $\mathcal{A}$ aggregates over m

Figure 4.6: Subsampling Transformation

parameters for each transformation to obtain the correct semantics for the mechanism.

Elastic Sensitivity

Let s be the Elastic Sensitivity of Q. Let $\gamma = s/\epsilon$. If $Q \xrightarrow{\gamma} Q'$, then Q' is an intrinsically private query via Elastic Sensitivity.

Restricted Sensitivity

Let s be the restricted sensitivity of Q. Let $\gamma = s/\epsilon$. If $Q \xrightarrow{\gamma} Q'$, then Q' is an intrinsically private query via Restricted Sensitivity.

Weighted PINQ

Let $i = 1$ and let j scale weights as described in Section 4.4. Let $\gamma = 1/\epsilon$. If $Q \overset{i,j,c}{\rightarrow} Q' \overset{\Gamma,\Gamma'}{\rightarrow} Q'' \xrightarrow{\gamma} Q'''$, then Q''' is an intrinsically private query via Weighted PINQ.

Sample & Aggregate

Let $\mathcal{A}$ implement private Winsorized mean [68] for a desired ϵ. If $Q \xrightarrow{\mathcal{A}} Q'$, then Q' is an intrinsically private query via Sample & Aggregate.

Correctness

The correctness criterion for a traditional query rewriting system is straightforward: the rewritten query should have the same semantics (i.e. return the same results) as the original query. For intrinsically private queries, however, the definition of correctness is less clear: enforcing differential privacy *requires* modifying the results to preserve privacy.

In this section, we define *semantic preservation*, which formalizes the intuitive notion that our transformations should perturb the result attributes of the query, as required for enforcing differential privacy, but change nothing else about its semantics. Semantic preservation holds when the transformation in question preserves the size and shape of the query's output, and additionally preserves its logical attributes. Logical attributes are those which are used as join keys or to perform

filtering (i.e. the query makes decisions based on these attributes, instead of simply outputting them).

Each of our transformations preserve this property. Furthermore, semantic preservation is preserved over composition of transformations, so semantic preservation holds for *any* mechanism defined using our transformations—including those defined earlier in this section.

Definition 8 (Logical Attribute)**.** *An attribute a is a logical attribute if it appears as a join key in a join expression, in the filter condition φ of a filter expression, or in the set of grouping attributes of a* Count *or* Sum *expression.*

Definition 9 (Semantic Preservation)**.** *A transformation ($\rightarrow$) satisfies* semantic preservation *if for all queries Q and Q', if $Q \rightarrow Q'$, then (1) the schema is preserved: $Q : U \Rightarrow Q' : U$; (2) the number of output rows is preserved: $|Q| = |Q'|$; and (3) logical attributes are preserved (see Definition 10).*

Definition 10 (Logical Attribute Preservation)**.** *Consider a transformation $Q : U \rightarrow Q' : U$. Split U into two sets of attributes $\{U_k, U_a\}$, such that U_k contains all of the attributes from U used as logical attributes in Q and U_a contains all of the other attributes. Now construct Q'_r by renaming each attribute $k \in U_k$ in Q' to k'. Then $\rightarrow$ preserves logical attributes if there exists a one-to-one relation $E \subseteq Q \times Q'_r$ such that for all $e \in E$ and $k \in U_k$, $e_k = e_{k'}$.*

Theorem 4 (Composition)**.** *If two transformations ($\rightarrow_a$ and $\rightarrow_b$) both satisfy semantic preservation, then their composition ($\rightarrow_c$) satisfies semantic preservation: for all queries Q, Q', and Q'', if $Q \rightarrow_a Q' \rightarrow_b Q''$ implies that $Q \rightarrow_c Q''$, and both $\rightarrow_a$ and $\rightarrow_b$ preserve semantics, then $\rightarrow_c$ preserves semantics.*

Proof. Assume that $\rightarrow_a$ and $\rightarrow_b$ preserve semantics, and $Q \rightarrow_a Q' \rightarrow_b Q''$. We have that: (1) $Q : U$, $Q' : U$, and $Q : U''$; (2) $|Q| = |Q'| = |Q''|$; and (3) Q, Q', and Q'' contain the same logical attributes. Thus by Definition 9, $\rightarrow_c$ preserves semantics. □

Theorem 5. *The Laplace Noise transformation (Figure 4.3) satisfies the semantic preservation property.*

Proof. The rules in Figure 4.3 define only one transformation, at the top level of the query. The Unif function adds a column u_x for each original column x; the Lap function consumes this column, replacing the original column x with its original value plus a value sampled from the Laplace distribution. The outermost projection produces exactly the same set of attributes as the input query Q, preserving the schema; the transformation adds only projection nodes, so the number of output rows is preserved; no join or select can occur at the top level of a query, so no logical attributes are modified. □

Theorem 6. *The Metadata Propagation transformation (Figure 4.4) satisfies the semantic preservation property.*

Proof. The only rules that make major modifications to the original query are those for joins and counts. The other rules add a new attribute m and propagate it through the query, but do not change the number or contents of the rows of any relation. At the top level of the query (i.e. $q \in Q$), the transformation eliminates the attribute m. For queries without joins or subquery aggregations, the Metadata Propagation transformation is the identity transformation, so it satisfies semantic preservation.

We argue the remaining cases by induction on the structure of Q.

Case $r_1 \underset{A=B}{\bowtie} r_2$. Let $r = r_1 \underset{A=B}{\bowtie} r_2$. If j does not change the query's semantics, except to update the attribute m (i.e. $r : U \Rightarrow r = \Pi_{U-m} j(r)$), then the semantics of r are preserved.

Case *Count*(r) and *Count*$_{G_1..G_n}(r)$. The rule modifies the m attribute, but does not modify any other attribute or change the size of the relation, so semantics are preserved. □

Theorem 7. *The Replace Aggregation Function transformation (Figure 4.5) satisfies the semantic preservation property.*

Proof. Aggregation function replacement has the potential to modify the values of the query's results, but not its shape or logical attributes. The transformation's only rule allows changing one function to another, but preserves the grouping columns and number of functions. The schema, number of rows, and logical attributes are therefore preserved. □

Theorem 8 (Subsampling preserves semantics)**.** *If the aggregation function $\mathcal{A}$ aggregates over the m attribute, then our Subsampling transformation (Figure 4.6) satisfies the semantic preservation property.*

Proof. We know that Q has the form *Count*(r) or $\underset{G_1..G_n,m}{\textit{Count}}\,(r)$. By Theorem 6, we know that in either case, if $r \overset{i,j,c}{\rightarrow} r'$, then r has the same semantics as r'. We proceed by cases.

Case $Q =$ *Count*(r). Let $q_1 =$ *Count*$_{\mathrm{m}}(r')$. By the definition of the transformation, $Q' = \mathcal{A}(q_1)$. The query q_1 has exactly one row per unique value of m. Since $\mathcal{A}$ aggregates over m, $\mathcal{A}(q_1)$ has exactly one row, and therefore preserves the semantics of Q.

Case $Q = \underset{G_1..G_n,m}{\textit{Count}}\,(r)$. Let $q_1 = \underset{G_1..G_n,m}{\textit{Count}}\,(r')$. By the definition of the transformation, $Q' = \mathcal{A}(q_1)$. The query q_1 has exactly one row per unique tuple $(G_1..G_n, \mathrm{m})$. Since $\mathcal{A}$ aggregates over m, $\mathcal{A}(q_1)$ has exactly one row per unique tuple $(G_1..G_n)$, and therefore preserves the semantics of Q.

□

Handling Histogram Queries

SQL queries can use the `GROUP BY` operator to return a relation representing a histogram, as in the following query which counts the number of trips greater than 100 miles in each city:

```
SELECT city_id, COUNT(*) as count FROM trips WHERE distance > 100
GROUP BY city_id
```

This type of query presents a problem for differential privacy because the presence or absence of a particular city in the results reveals whether the count for that city was zero.

The general solution to this problem is to require the analyst to explicitly provide a set of desired bins, and return a (noisy) count of zero for absent bins. Such an approach is used, for example, in PINQ [55], Weighted PINQ [63], and FLEX (Chapter 3). Unfortunately, this approach imposes an unfamiliar user experience and is burdensome for the analyst, who is never allowed to view the results directly.

For bins with finite domain, CHORUS provides a superior solution by enumerating missing histogram bins automatically *in the rewritten query*. The missing bins are populated with empty aggregation values (e.g., 0 for counts) before mechanism-specific rewriting, at which point they are handled identically as non-empty bins. This allows the full results of histogram queries to be returned to the analyst without post-processing or interposition. If the domain cannot be enumerated (e.g., because it is unbounded), CHORUS falls back to the approach described above and does not release results directly to the analyst.

This feature requires the operator to define a mapping from columns that may be used in a `GROUP BY` clause to the database field containing the full set of values from that column's domain. This information may be defined manually or extracted from the database schema (e.g., via primary and foreign key constraints), and is provided during initial deployment.

In this example, suppose the full set of city ids are stored in column `city_id` of table `cities`. Using this information, CHORUS generates the following intermediate query:

```
WITH orig AS (
  SELECT city_id, COUNT(*) as count FROM trips
  WHERE distance > 100
  GROUP BY city_id
)
SELECT cities.city_id,
       (CASE WHEN orig.count IS NULL THEN 0
             ELSE orig.count END) as count
FROM orig RIGHT JOIN cities
  ON orig.city_id = cities.city_id
```

The `RIGHT JOIN` ensures that exactly one row exists in the output relation for each city_id in `cities`, and the `CASE` expression outputs a zero for each missing city in the original query's results. The query thus contains *every* city_id value regardless of the semantics of the original query. This intermediate query is then sent to the mechanism rewriter, which adds noise to each bin as normal.

4.6 Implementation

This section describes our implementation of CHORUS and our experience deploying it to enforce differential privacy at Uber.

Implementation

Our implementation of CHORUS automatically transforms an input SQL query into an intrinsically private query. CHORUS currently supports the four differential privacy mechanisms discussed here, and is designed for easy extension to new mechanisms. We have released CHORUS as an open source project [3].

Our implementation is built on Apache Calcite [1], a generic query optimization framework that transforms input queries into a relational algebra tree and provides facilities for transforming the tree and emitting a new SQL query. We built a custom dataflow analysis and rewriting framework on Calcite to support intrinsically private queries. The framework, mechanism-specific analyses, and rewriting rules are implemented in 5,096 lines of Java and Scala code.

The approach could also be implemented with other query optimization frameworks or rule-based query rewriters such as Starburst [62], ORCA [7], and Cascades [39].

Privacy Budget Management

We have designed CHORUS to be flexible in its handling of the privacy budget, since best approach in a given setting is likely to depend on the domain and the kinds of queries posed. A complete study of approaches for managing the privacy budget is beyond the scope of this work, but we outline some possible strategies here. We describe the specific method used in our deployment in the next subsection.

As described earlier, a simple approach to budget management involves standard composition. More sophisticated methods for privacy budget accounting include the *advanced composition* [36] and *parallel composition* [32], both of which are directly applicable in our setting. For some mechanisms, the moments account [17] could be used to further reduce privacy budget depletion.

Support for other mechanisms. Mechanisms themselves can also have a positive effect on the privacy budget, and many mechanisms have been designed to provide improved accuracy for a workload of similar queries. Many of these mechanisms are implemented in terms of lower-level mechanisms (such as the Laplace mechanism) that CHORUS already supports, and therefore could be easily integrated in CHORUS.

The *sparse vector technique* [34] answers a sequence of queries, but only gives answers for queries whose results lie above a given threshold. The technique is implemented in terms of the Laplace mechanism.

The *Matrix Mechanism* [51] and *MWEM* [41] algorithms both answer a query workload by posing carefully chosen queries on the private database using a lower-level mechanism (e.g. the Laplace mechanism), then using the results to build a representation that can answer the queries in the workload.

The *Exponential Mechanism* [54] enforces differential privacy for queries that return categorical (rather than numeric) data, by picking from the possible outputs with probability generated from an analyst-provided *scoring function*. This technique can be be implemented as an intrinsically private query if the scoring function is given in SQL; the transformed query can run the

function on each possible output and then pick from the possibilities according to the generated distribution.

Deployment

CHORUS is currently deployed at Uber to enforce differential privacy for internal data access. The primary goals of this deployment are to protect the privacy of customers while enabling ad-hoc data analytics, and to ensure compliance with the requirements of Europe's General Data Protection Regulation (GDPR) [6]. In the current deployment, CHORUS processes more than 10,000 queries per day.

Data environment & architecture. The data environment into which CHORUS is deployed consists of several DBMSs (three primary databases, plus several more for specific applications), and a single central query interface through which all queries are submitted. The query interface is implemented as a microservice that performs query processing and then submits the query to the appropriate DBMS and returns the results.

Our deployment involves a minimal wrapper around the CHORUS library to expose its rewriting functionality as a microservice. The only required change to the data environment was a single modification to the query interface, to submit queries to the CHORUS microservice for rewriting before execution. The wrapper around CHORUS also queries a policy microservice to determine the security and privacy policy for the user submitting the query. This policy informs which rewriter is used—by default, differential privacy is required, but for some privileged users performing specific business tasks, differential privacy is only used for older data.

A major challenge of this deployment has been supporting the variety of SQL dialects used by the various DBMSs. The Calcite framework is intended to provide support for multiple dialects, but this support is incomplete and we have had to make changes to Calcite in order to support custom SQL dialects such as Vertica.

Privacy budget. The privacy budget is managed by the microservice wrapper around CHORUS. The microservice maintains a small amount of state to keep track of the current cumulative privacy cost of all queries submitted so far, and updates this state when a new query is submitted.

The current design of the CHORUS microservice maintains a single global budget, and uses advanced composition [35] to track the total budget used for the queries submitted so far.

As we gain experience with the deployment, we are beginning to consider more sophisticated budget management approaches that take advantages of properties of the data and the query workload. For example, new data is added to the database continuously so recent work leveraging the growth of the database to answer an unbounded number of queries [26] may be directly applicable.

Mechanism selection. Our deployment of CHORUS leverages a syntax-based selection procedure which aims to optimize for utility (low error). As we show in Section 4.7, this approach

performs well for this deployment. For different query workloads, other approaches may work significantly better, and CHORUS is designed to support extension to these cases.

The syntax-based approach uses a set of rules that map SQL constructs supported by each mechanism to a heuristic scoring function indicating how likely queries using that construct will yield high utility. For example, Restricted sensitivity [20] supports counting queries with joins, but does not support many-to-many joins. Elastic sensitivity [45] supports a wider set of equi-joins, including many-to-many joins, but generally provides slightly lower utility than Restricted sensitivity due to smoothing. Sample & aggregate [68] does not support joins, but does support additional aggregation functions (including average and median).

When a query is submitted, the mechanism selection engine analyzes the query to determine its syntactic properties including how many and what types of joins it has, and what aggregation functions it uses. It then applies the rules to determine which mechanisms can support the query and selects the mechanism with highest score. Since this process does not depend on the data it does not consume privacy budget.

This approach represents a simple but effective strategy for automatic mechanism selection. In Section 4.7, we demonstrate that our rules are effective for selecting the best mechanism on a real-world query workload. This approach is also easily extended when a new mechanism is added: the mechanism designer simply adds new rules for SQL constructs supported by the mechanism. Moreover, the scoring function can be tuned for other objectives, for example favoring mechanisms achieving low performance overhead rather than highest utility.

4.7 Evaluation

This section reports results of the following experiments:

- We report the percentage of queries that can be supported by each mechanism as intrinsically privacy queries using a corpus of real-world queries, demonstrating that a combination of the four evaluated mechanisms covers 93.9% of these queries.
- We use CHORUS to conduct the first empirical study of several differential privacy mechanisms on a real-world SQL workload. We report the performance overhead and utility of each mechanism across its supported class of queries.
- We demonstrate that our rule-based approach for automatic mechanism selection is effective at selecting the best mechanism for each input query. Using the simple set of rules presented earlier, our approach selects the optimal mechanism for nearly 90% of the queries in our corpus.

Corpus. We use a corpus of 18,774 real-world queries containing all statistical queries executed by data analysts at Uber during October 2016.

The corpus includes queries written for several use cases including fraud detection, marketing, business intelligence and general data exploration. It is therefore highly diverse and representative

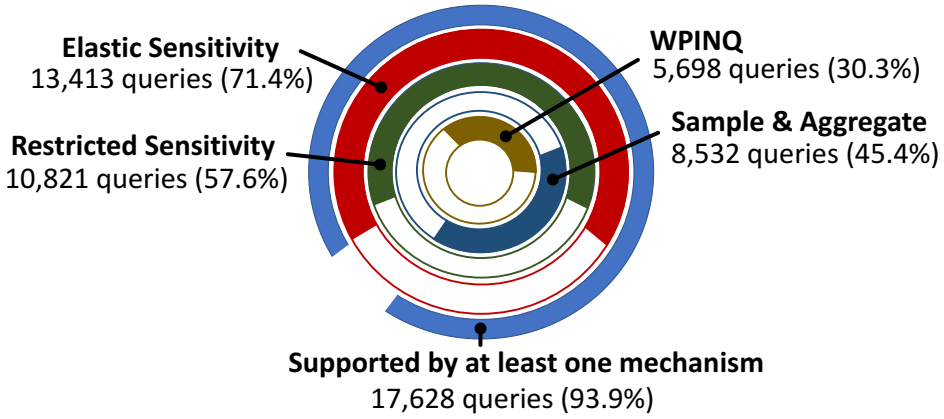


Figure 4.7: Size and relationship of query sets supported by each evaluated mechanism.

of SQL data analytics queries generally. The queries were executed on a database of data sampled from the production database.

Mechanism Support for Queries

Each mechanism evaluated supports a different subset of queries. This is due to the unique limitations and supported constructs of each mechanism, as summarized in Section 4.3. We measured the percentage of queries from our corpus supported by each mechanism to assess that mechanism's coverage on a real-world workload. Figure 4.7 depicts the relative size and overlap of each set of supported queries for the evaluated mechanisms. Elastic Sensitivity is the most general mechanism and can support 71.4% of the queries in our corpus, followed by Restricted Sensitivity (57.6%), WPINQ (30.3%) and Sample & Aggregate (45.4%).

Elastic Sensitivity and Restricted Sensitivity support largely the same class of queries, and WPINQ supports a subset of the queries supported by these two mechanisms. In Section 4.7 we discuss limitations preventing the use of WPINQ for certain classes of queries supported by Elastic Sensitivity and Restricted Sensitivity.

Sample & Aggregate supports some queries supported by other mechanisms (counting queries that do not use join), as well as a class of queries using statistical estimators (such as sum and average), that are not supported by the other mechanisms. In total, 93.9% of queries are supported by at least one of the four mechanisms.

The results highlight a key advantage of our approach: different classes of queries can be simultaneously supported via selection of one or more specialized mechanisms. This ensures robust support across a wide range of general and specialized use cases, and allows incremental adoption of future state-of-the-art mechanisms.

	Overhead (%)		Primary cause
	Mean	Median	of overhead
Elastic Sensitivity	2.8	1.7	Random noise generation
Restricted Sensitivity	3.2	1.6	Random noise generation
WPINQ	50.9	21.6	Additional joins
Sample & Aggregate	587	394	Grouping/aggregation

Table 4.2: Performance overhead of evaluated differential privacy mechanisms.

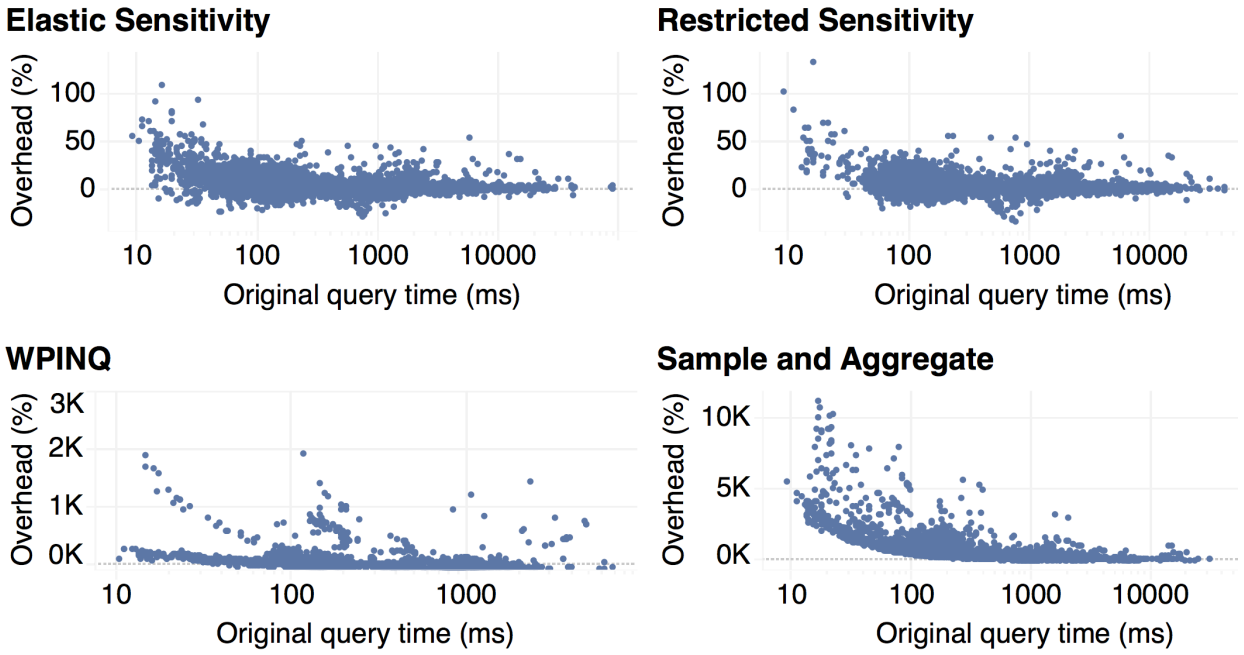


Figure 4.8: Performance overhead of differential privacy mechanisms by execution time of original query.

Performance Overhead

We conduct a performance evaluation demonstrating the performance overhead of each mechanism when implemented as an intrinsically private query.

Experiment Setup. We used a single HP Vertica 7.2.3 [10] node containing 300 million records including trips, rider and driver information and other associated data stored across 8 tables. We submitted the queries locally and ran queries sequentially to avoid any effects from network latency and concurrent workloads.

To establish a baseline we ran each original query 10 times and recorded the average after dropping the lowest and highest times to control for outliers. For every mechanism, we used CHORUS to transform each of the mechanism's supported queries into an intrinsically private query. We

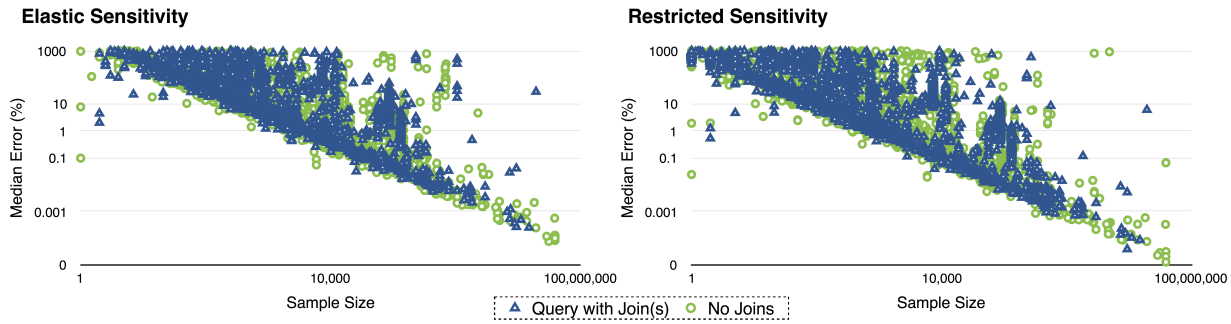


Figure 4.9: Utility of elastic sensitivity and restricted sensitivity, by presence of joins.

executed each intrinsically private query 10 times and recorded the average execution time, again dropping the fastest and slowest times. We calculate the overhead for each query by comparing the average runtime of the intrinsically private query against its baseline.[4]

Results. The results are presented in Table 4.2. The average overhead and median overhead for Elastic Sensitivity are 2.82% and 1.7%, for Restricted Sensitivity these are 3.2% and 1.6%, for WPINQ 50.9% and 21.58% and for Sample & Aggregate 587% and 394%.

Figure 4.9 shows the distribution of overhead as a function of original query execution time. This distribution shows that the percentage overhead is highest when the original query was very fast (less than 100ms). This is because even a small incremental performance cost is fractionally larger for these queries. The values reported in Table 4.2 are therefore a conservative estimate of the overhead apparent to the analyst.

WPINQ and Sample & Aggregate significantly alter the way the query executes (see Section 4.4) and these changes increase query execution time. In the case of WPINQ, the query transformation adds a new join to the query each time weights are rescaled (i.e. one new join for each join in the original query), and these new joins produce additional overhead. Sample & Aggregate requires additional grouping and aggregation steps. We hypothesize that these transformations are difficult for the database to optimize during execution. Figure 4.9 shows that, in both cases, the performance impact is amortized over higher query execution times, resulting in a lower relative overhead for more expensive queries.

Utility of Selected Mechanisms

CHORUS enables the first empirical study of the *utility* of many differential privacy mechanisms on a real-world query workload. This experiment reveals innate trends of each mechanism on a common database and query workload. For each differential privacy mechanism, this experiment reports the relative magnitude of error added to results of its supported query set. We present the results as a function of query sample size, discussed below.

[4]Transformation time is negligible and therefore not included in the overhead calculation. The transformation time averages a few milliseconds, compared with an average query execution time of 1.5 seconds.

Experiment Setup. We use the same setup described in the previous section to evaluate the utility of Elastic Sensitivity, Restricted Sensitivity, and Sample & Aggregate. As described in Section 4.4, WPINQ's output is a differentially private statistic used as input to a post-processing step, rather than a direct answer to the query. The authors [63] do not describe how to automate this step for new queries hence we cannot measure the utility of WPINQ directly.

For each query, we set the privacy budget $\epsilon = 0.1$ for all mechanisms. For Elastic Sensitivity, we set $\delta = n^{-\epsilon \ln n}$ (where n is the database size), following Dwork and Lei [31]. For Sample & Aggregate, we set the number of subsamples $\ell = n^{0.4}$, following Mohan et al. [56].

We ran each intrinsically private query 10 times on the database and report the median relative error across these executions. For each run we report the relative error as the percentage difference between the differentially private result and the original non-private result. Consistent with previous evaluations of differential privacy [43] we report error as a proxy for utility since data analysts are primarily concerned with accuracy of results.

If a query returns multiple rows (e.g., histogram queries) we calculate the mean error across all histogram bins. If the query returns multiple columns we treat each output column independently since noise is applied separately to every column.

Query Sample Size. Our corpus includes queries covering a broad spectrum of use cases, from highly selective analytics (e.g., trips in San Francisco completed in the past hour) to statistics of large populations (e.g., all trips in the US). Differential privacy generally requires the addition of more noise to highly selective queries than to queries over large populations, since the influence of any individual's data diminishes as population size increases. Consequently, a query's selectivity is important for interpreting the relative error introduced by differential privacy. To measure the selectivity we calculate the *sample size* of every aggregation function in the original query, which represents the number of input records to which the function was applied.

Results. Figures 4.9 and 4.10 show the results of this experiment. All three mechanisms exhibit the expected inverse relationship between sample size and error; moreover, this trend is apparent for queries with and without joins.

Where the other three mechanisms support only counting queries, Sample & Aggregate supports all statistical estimators. Figure 4.10 shows the utility results for Sample & Aggregate, highlighting the aggregation function used. These results indicate that Sample & Aggregate can provide high utility ($<$10% error) for each of its supported aggregation functions on approximately half of the queries.

Automatic Mechanism Selection

We evaluated the effectiveness of the syntax-based automatic mechanism selection approach described in Section 4.6. For each query in our corpus, this experiment compares the utility achieved by the mechanism selected by our rule-based approach to the best possible utility achievable by any mechanism implemented in CHORUS.

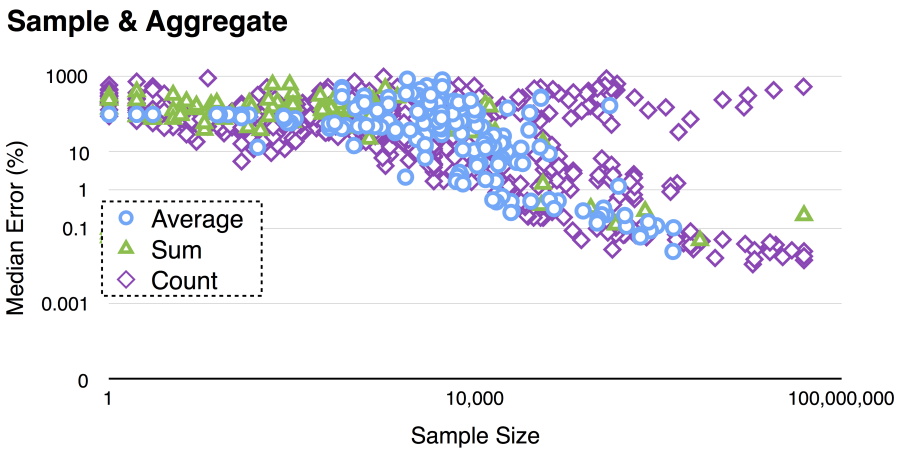


Figure 4.10: Utility of Sample & Aggregate, by aggregation function.

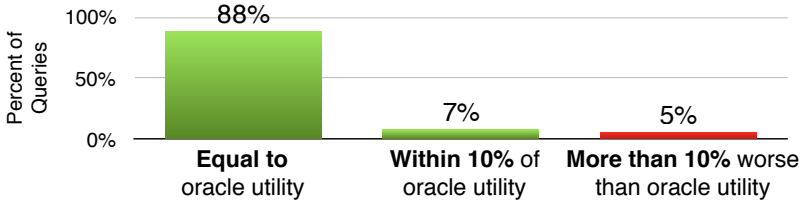


Figure 4.11: Effectiveness of automatic mechanism selection.

Experiment Setup. We used the same corpus of queries and the same database of trips as in the other experiments. For each query, we ran all of the mechanisms that support the query and recorded the relative error (i.e. utility) of each one. We defined the *oracle utility* for each query to be the minimum error achieved by any of the four implemented mechanisms for that query. The oracle utility is intended to represent the utility that could be obtained if a perfect oracle for mechanism selection were available. We used our syntax-based mechanism selection method to select a single mechanism, and determined the utility of that mechanism. Finally, we computed the difference between the oracle utility and the utility achieved by our selected mechanism.

Results. We present the results in Figure 4.11. For 88% of the queries in our corpus, the automatic mechanism selection rules select the best mechanism, and therefore provide the same utility as the oracle utility. Of the remaining queries, the selected mechanism often provides nearly optimal utility: for 7% of queries, the selected mechanism is within 10% error of the oracle utility.

The remaining queries (5%) represent opportunities for improving the approach—perhaps through the use of a prediction model trained on features of the query and data. Previous work [43, 49] uses such a machine learning-based approach; for range queries, these results suggest that a learning-based approach can be very effective, though the approach has not been evaluated on other types of queries.

Discussion and Key Takeaways

Strengths & weaknesses of differential privacy. The mechanisms we studied generally worked best for statistical queries over large populations. None of the mechanisms was able to provide accurate results (e.g. within 1% error) for a significant number of queries over populations smaller than 1,000. These results confirm the existing wisdom that differential privacy is ill-suited for queries with small sample sizes. For large populations (e.g. more than 10,000), on the other hand, multiple mechanisms were able to provide accurate results.

Mechanism performance. Our performance evaluation highlights the variability in computation costs of differential privacy mechanisms. Approaches such as Elastic Sensitivity or Restricted Sensitivity incur little overhead, suggesting these mechanisms are ideal for performance critical applications such as real-time analytics. Given their higher performance cost, mechanisms such as WPINQ and Sample & Aggregate may be most appropriate for specialized applications where performance is less important than suitability of the mechanism for a particular problem domain. For example, WPINQ is the only evaluated mechanism that supports synthetic data generation, a task known to be highly computation-intensive.

The performance of intrinsically private queries can depend on the database engine and transformations applied to the query. In this work we do not attempt to optimize the rewritten queries for performance.

Unsupported queries. The current implementation of CHORUS applies a single mechanism to an entire input query. As a result, every aggregation function used by the input query must be supported by the selected mechanism, or the transformation fails. For example, consider a query with joins that outputs both a count and an average. Neither Elastic Sensitivity (which does not support average) nor Sample & Aggregate (which does not support join) can fully support this query.

This issue disproportionately affects WPINQ, since our implementation of WPINQ does not support `COUNT(DISTINCT ...)` queries. It is not obvious how to do so: the weights of any record in the database only reflect the number of duplicate rows until a join is performed (and weights are re-scaled).

It is possible to leverage multiple mechanisms in a single intrinsically private query by treating each output column separately. This approach would provide improved support for queries like the example above, which use several different aggregation functions. We leave such an extension to future work.

4.8 Related Work

Differential Privacy. Differential privacy was originally proposed by Dwork [29, 30, 32]. The reference by Dwork and Roth [35] provides an overview of the field.

Much recent work has focused on task-specific mechanisms for graph analysis [23, 42, 46, 47, 66], range queries [18, 24, 41, 50–52, 64, 71–73], and analysis of data streams [33, 67]. As described in Section 4.7, such mechanisms are complementary to our approach, and could be implemented on top of CHORUS to provide more efficient use of the privacy budget.

Differential Privacy Systems. A number of systems for enforcing differential privacy have been developed. PINQ [55] supports a LINQ-based query language, and implements the Laplace mechanism with a measure of global sensitivity. Weighted PINQ [63] extends PINQ to weighted datasets, and implements a specialized mechanism for that setting.

Airavat [65] enforces differential privacy for MapReduce programs using the Laplace mechanism. Fuzz [38, 40] enforces differential privacy for functional programs, using the Laplace mechanism in an approach similar to PINQ. DJoin [57] enforces differential privacy for queries over distributed datasets. Due to the additional restrictions associated with this setting, DJoin requires the use of special cryptographic functions during query execution so is incompatible with existing databases. GUPT [56] implements the Sample & Aggregate framework for Python programs.

In contrast to our approach, each of these systems supports only a single mechanism and, with the exception of Airavat, each implements a custom database engine.

Security & Privacy via Query Rewriting. To the best of our knowledge, ours is the first work on using query transformations to implement differential privacy mechanisms. However, this approach has been used in previous work to implement access control. Stonebreaker and Wong [69] presented the first approach. Barker and Rosenthal [19] extended the approach to role-based access control by first constructing a view that encodes the access control policy, then rewriting input queries to add `WHERE` clauses that query the view. Byun and Li [22] use a similar approach to enforce purpose-based access control: purposes are attached to data in the database, then queries are modified to enforce purpose restrictions drawn from a policy.

Chapter 5

Conclusion

This dissertation takes a first step toward practical differential privacy for general-purpose SQL queries in real-world environments.

We conducted the largest known empirical study of real-world SQL queries—8.1 million queries in total—and from these results proposed a new set of requirements for practical differential privacy on SQL queries. To meet these requirements we proposed elastic sensitivity, the first efficiently-computed approximation of local sensitivity that supports joins.

We then presented CHORUS, a system with a novel architecture based on query rewriting that enforces differential privacy for SQL queries on an unmodified database. CHORUS works by automatically transforming input queries into intrinsically private queries. We have described transformation rules for four general-purpose mechanisms, and discussed how additional mechanisms can be supported with the same approach.

We used CHORUS to perform the first empirical evaluation of multiple mechanisms on real-world queries and data. The results demonstrate that our approach supports 93.9% of statistical queries in our corpus, integrates with a production DBMS without any modifications, and scales to hundreds of millions of records.

We described the deployment of CHORUS at Uber to provide differential privacy, where it processes more than 10,000 queries per day. We have released CHORUS as open source [3].

As organizations increasingly collect sensitive information about users, they are highly motivated to make the data available to analysts in order to maximize its value. At the same time, data breaches are becoming more common and the public is increasingly concerned about privacy. There is a growing and urgent need for technology solutions that balance these interests by supporting general-purpose analytics while simultaneously providing privacy protection.

Differential privacy is a promising solution to this problem that enables general data analytics while protecting individual privacy, but existing differential privacy mechanisms do not support the wide variety of features and databases of real-world environments.

A flexible, practical system that supports multiple state-of-the-art mechanisms could accelerate adoption of differential privacy in practice. In addition, the ability to evaluate current and future mechanisms in a real-world setting will support development of new mechanisms with greater utility and expand the application domain of differential privacy.

Bibliography

[1] Apache Calcite. http://calcite.apache.org. Accessed: 2017-04-27.

[2] Check Constraint. https://msdn.microsoft.com/en-us/library/ms190377.aspx.

[3] Chorus Source Code. https://github.com/uber/sql-differential-privacy.

[4] Database Triggers. https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch15.htm.

[5] Elastic Sensitivity experiments using TPC-H. http://www.github.com/sunblaze-ucb/elastic-sensitivity-experiments.

[6] General Data Protection Regulation - Wikipedia. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[7] GPORCA. http://engineering.pivotal.io/post/gporca-open-source/. Accessed: 2017-04-27.

[8] High Energy Physics - Theory collaboration network. https://snap.stanford.edu/data/ca-HepTh.html.

[9] Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io/.

[10] Vertica Advanced Analytics. http://www8.hp.com/us/en/software-solutions/advanced-sql-big-data-analytics/. Accessed: 2017-01-20.

[11] Weighted Privacy Integrated Queries. http://cs-people.bu.edu/dproserp/wPINQ.html.

[12] Swiss spy agency warns U.S., Britain about huge data leak. *Reuters World News*, December 4, 2012. http://www.reuters.com/article/us-usa-switzerland-datatheft-idUSBRE8B30ID20121204.

[13] Nearly 5,000 patients affected by UC Irvine medical data breach. *Los Angeles Times*, June 18, 2015. http://www.latimes.com/business/la-fi-uc-irvine-data-breach-20150618-story.html.

[14] Sutter Health California Pacific Medical Center audit uncovers data breach. *CSO*, January 28, 2015. `http://www.csoonline.com/article/2876324/data-breach/sutter-health-california-pacific-medical-center-audit-uncovers-data-breach.html`.

[15] Apple previews iOS 10, the biggest iOS release ever. *Apple Press Release*, June 13, 2016. `http://www.apple.com/newsroom/2016/06/apple-previews-ios-10-biggest-ios-release-ever.html`.

[16] Morgan Stanley Breach a Reminder of Insider Risks. *Security Intelligence*, January 8, 2016. `https://securityintelligence.com/news/morgan-stanley-breach-reminder-insider-risks/`.

[17] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[18] G. Acs, C. Castelluccia, and R. Chen. Differentially private histogram publishing through lossy compression. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 1–10. IEEE, 2012.

[19] S. Barker and A. Rosenthal. Flexible security policies in sql. In *Database and Application Security XV*, pages 167–180. Springer, 2002.

[20] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 87–96, New York, NY, USA, 2013. ACM.

[21] J. Blocki, A. Datta, and J. Bonneau. Differentially private password frequency lists. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[22] J.-W. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal*, 17(4):603–619, 2008.

[23] S. Chen and S. Zhou. Recursive mechanism: Towards node differential privacy and unrestricted joins. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 653–664, New York, NY, USA, 2013. ACM.

[24] G. Cormode, C. Procopiuc, D. Srivastava, E. Shen, and T. Yu. Differentially private spatial decompositions. In *Data engineering (ICDE), 2012 IEEE 28th international conference on*, pages 20–31. IEEE, 2012.

[25] T. P. P. Council. Tpc-h benchmark specification. *Published at `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf`*, 21:592–603, 2008.

[26] R. Cummings, S. Krehbiel, K. A. Lai, and U. Tantipongpipat. Differential privacy for growing databases. *arXiv preprint arXiv:1803.06416*, 2018.

[27] Y.-A. de Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3, 2013.

[28] B. Ding, M. Winslett, J. Han, and Z. Li. Differentially private data cubes: optimizing noise sources and consistency. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 217–228. ACM, 2011.

[29] C. Dwork. Differential privacy. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2006.

[30] C. Dwork. Differential privacy: A survey of results. In *Theory and applications of models of computation*, pages 1–19. Springer, 2008.

[31] C. Dwork and J. Lei. Differential privacy and robust statistics. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 371–380. ACM, 2009.

[32] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[33] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 715–724. ACM, 2010.

[34] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. Vadhan. On the complexity of differentially private data release: efficient algorithms and hardness results. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 381–390. ACM, 2009.

[35] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[36] C. Dwork, G. N. Rothblum, and S. Vadhan. Boosting and differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 51–60. IEEE, 2010.

[37] Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067. ACM, 2014.

[38] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN Notices*, volume 48, pages 357–370. ACM, 2013.

[39] G. Graefe. The cascades framework for query optimization.

[40] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *USENIX Security Symposium*, 2011.

[41] M. Hardt, K. Ligett, and F. McSherry. A simple and practical algorithm for differentially private data release. In *Advances in Neural Information Processing Systems*, pages 2339–2347, 2012.

[42] M. Hay, C. Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 169–178. IEEE, 2009.

[43] M. Hay, A. Machanavajjhala, G. Miklau, Y. Chen, and D. Zhang. Principled evaluation of differentially private algorithms using dpbench. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 139–154. ACM, 2016.

[44] X. Hu, M. Yuan, J. Yao, Y. Deng, L. Chen, Q. Yang, H. Guan, and J. Zeng. Differential privacy in telco big data platform. *PVLDB*, 8(12):1692–1703, 2015.

[45] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proc. VLDB Endow.*, 11(5):526–539, Jan. 2018.

[46] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. *PVLDB*, 4(11):1146–1157, 2011.

[47] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Analyzing graphs with node differential privacy. In *Theory of Cryptography*, pages 457–476. Springer, 2013.

[48] D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 193–204. ACM, 2011.

[49] I. Kotsogiannis, A. Machanavajjhala, M. Hay, and G. Miklau. Pythia: Data dependent differentially private algorithm selection. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1323–1337. ACM, 2017.

[50] C. Li, M. Hay, G. Miklau, and Y. Wang. A data-and workload-aware algorithm for range queries under differential privacy. *PVLDB*, 7(5):341–352, 2014.

[51] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 123–134. ACM, 2010.

[52] C. Li, G. Miklau, M. Hay, A. McGregor, and V. Rastogi. The matrix mechanism: optimizing linear counting queries under differential privacy. *The VLDB Journal*, 24(6):757–781, 2015.

[53] W. Lu, G. Miklau, and V. Gupta. Generating private synthetic databases for untrusted system evaluation. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 652–663. IEEE, 2014.

[54] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 94–103. IEEE, 2007.

[55] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30. ACM, 2009.

[56] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. Gupt: privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2012.

[57] A. Narayan and A. Haeberlen. Djoin: differentially private join queries over distributed databases. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 149–162, 2012.

[58] A. Narayanan and V. Shmatikov. How to break anonymity of the Netflix prize dataset. *CoRR*, abs/cs/0610105, 2006.

[59] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 75–84. ACM, 2007.

[60] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. 2011. Draft Full Version, v1.0. `http://www.cse.psu.edu/~ads22/pubs/NRS07/NRS07-full-draft-v1.pdf`.

[61] V. Pandurangan. On taxis and rainbows: Lessons from NYC's improperly anonymized taxi logs. https://medium.com/@vijayp/of-taxis-and-rainbows-f6bc289679a1.

[62] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 39–48, New York, NY, USA, 1992. ACM.

[63] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets. *PVLDB*, 7(8):637–648, 2014.

[64] W. Qardaji, W. Yang, and N. Li. Differentially private grids for geospatial data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 757–768. IEEE, 2013.

[65] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, volume 10, pages 297–312, 2010.

[66] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 81–98. ACM, 2011.

[67] E. Shi, H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium (NDSS)*. Internet Society., 2011.

[68] A. Smith. Privacy-preserving statistical estimation with optimal convergence rates. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 813–822. ACM, 2011.

[69] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 annual conference-Volume 1*, pages 180–186. ACM, 1974.

[70] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *The Journal of Law, Medicine & Ethics*, 25(2-3):98–110, 1997.

[71] Y. Xiao, L. Xiong, L. Fan, and S. Goryczka. Dpcube: differentially private histogram release through multidimensional partitioning. *arXiv preprint arXiv:1202.5358*, 2012.

[72] J. Xu, Z. Zhang, X. Xiao, Y. Yang, G. Yu, and M. Winslett. Differentially private histogram publication. *The VLDB Journal*, 22(6):797–822, 2013.

[73] X. Zhang, R. Chen, J. Xu, X. Meng, and Y. Xie. Towards accurate histogram publication under differential privacy. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 587–595. SIAM, 2014.