Towards Practical Differential Privacy for SQL Queries

Noah Johnson
University of California, Berkeley
noahj@berkeley.edu

Joseph P. Near
University of California, Berkeley
jnear@berkeley.edu

Dawn Song
University of California, Berkeley
dawnsong@cs.berkeley.edu

arXiv:1706.09479v2 [cs.CR] 16 Aug 2017

ABSTRACT

Differential privacy promises to enable general data analytics while protecting individual privacy, but existing differential privacy mechanisms do not support the wide variety of features and databases used in real-world SQL-based analytics systems.

This paper presents the first practical approach for differential privacy of SQL queries. Using 8.1 million real-world queries, we conduct an empirical study to determine the requirements for practical differential privacy, and discuss limitations of previous approaches in light of these requirements. To meet these requirements we propose elastic sensitivity, 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 can therefore be used to enforce differential privacy using any local sensitivity-based mechanism.

We build FLEX, a practical end-to-end system to enforce differential privacy for SQL queries using elastic sensitivity. We demonstrate that FLEX is compatible with any existing database, can enforce differential privacy for real-world SQL queries, and incurs negligible (0.03%) performance overhead.

1. INTRODUCTION

As organizations increasingly collect sensitive information about individuals, these organizations are ethically and legally obligated to safeguard against privacy leaks. Data analysts within these organizations, however, have come to depend on unrestricted access to data for maximum productivity. This access is frequently provided in the form of a relational database that supports SQL queries. Current approaches for data security and privacy cannot guarantee privacy for individuals while providing general-purpose access for the analyst.

As demonstrated by recent insider attacks [7, 8, 10, 11], allowing members of an organization unrestricted access to data is a major cause of privacy breaches. 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 [18, 43, 45, 50].

Differential privacy [20, 23, 25] is a promising technique for addressing these issues. Differential privacy allows general statistical analysis of data while protecting *data about individuals* with a strong formal guarantee of privacy.

Because of its desirable formal guarantees, differential privacy has received growing attention from organizations including Google and Apple. However, research on practical techniques for differential privacy has focused on special-purpose use cases, such as collecting statistics about web browsing behaviors [27] and keyboard and emoji use [1], while differential privacy for general-purpose data analytics remains an open challenge.

Various mechanisms [14, 40–42, 44, 46] provide differential privacy for some subsets of SQL-like queries, but none support the majority of queries in practice. These mechanisms also require modifications to the database engine, complicating adoption in practice.

Furthermore, 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 differential privacy mechanisms on real-world data [15,32,33], however this work uses a limited set of queries representing a single, special-purpose analytics task such as histogram analysis [15] or range queries [32]. To the best of our knowledge, no existing work has explored the design and evaluation of differential privacy techniques for general, real-world queries.

This paper proposes *elastic sensitivity*, a novel approach for differential privacy of SQL queries. In contrast to existing work, our approach is compatible with real database systems, supports queries expressed in standard SQL, and integrates easily into existing data environments. The work therefore represents a first step towards practical differential privacy. The approach has recently been adopted by Uber to enforce differential privacy for internal data analytics [12].

We developed elastic sensitivity using requirements derived from a dataset of 8.1 million real-world queries. This paper focuses on counting queries, which constitute the majority of statistical queries in this dataset, and discusses extensions to the approach for other aggregation functions. We have released an open-source tool for computing elastic sensitivity of SQL queries [4].

Contributions. We make four primary contributions toward practical differential privacy:

1. We conduct the largest known empirical study of real-world SQL queries—8.1 million queries in total. From these results we show that the queries used in prior work to evaluate differential privacy mechanisms are not representative of real-world queries. We propose a new set of requirements for practical differential privacy on SQL queries based on these results.
2. To meet these requirements, we propose *elastic sensitivity*, a

sound approximation of local sensitivity [22, 44] 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.

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 database, can enforce differential privacy for the majority of real-world SQL queries, and incurs negligible (0.03%) performance overhead.
4. In the first experimental evaluation of its kind, we use FLEX to evaluate the impact of differential privacy on 9862 real-world statistical queries in our dataset. In contrast to previous empirical evaluations of differential privacy, our experimental set 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.

The rest of the paper is organized as follows. Section 2 contains our empirical study and defines the requirements for a practical differential privacy mechanism. In Section 3, we define elastic sensitivity and prove that it is an upper bound on local sensitivity. In Section 4 we describe FLEX, our system for enforcing differential privacy using elastic sensitivity. Section 5 contains our experimental evaluation of FLEX and Section 6 surveys related work.

2. REQUIREMENTS FOR PRACTICAL DIFFERENTIAL PRIVACY

We use a dataset consisting of millions of SQL queries to establish requirements for a practical differential privacy system that supports the majority of real-world queries. We investigate the limitations of existing general-purpose differential mechanisms in light of these requirements, and introduce *elastic sensitivity*, our new approach that meets these requirements.

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 many are used by 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.

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.

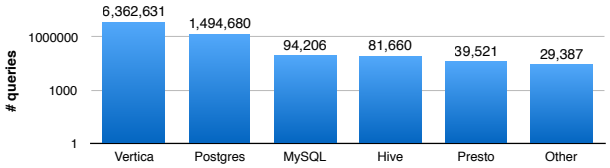
Data analysts at Uber query this information in support of many business interests such as improving service, detecting fraud, and understanding trends in the business. The majority of these use-cases require flexible, general-purpose analytics.

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

2.1 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?



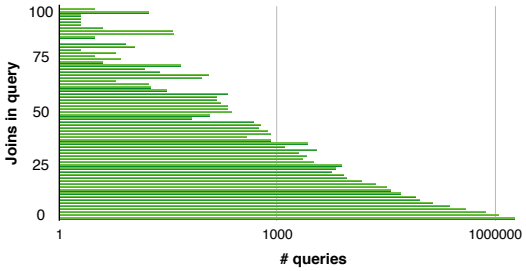
Results. The queries in our dataset use more than 6 database backends, including Vertica, Postgres, MySQL, Hive, and Presto.

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%

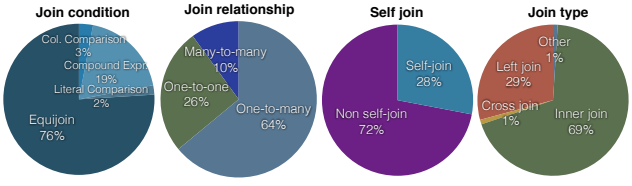
Results. All queries in our dataset use the `Select` operator, more than half of the queries use the `Join` operator, and fewer than 1 percent use other operators such as `Union`, `Minus`, and `Intersect`.

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



Results. A significant number of queries use multiple joins, with 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.

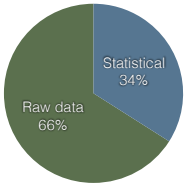
Compound expressions, defined as join conditions using function applications and conjunctions and disjunctions of primitive operators, account for 19% of join conditions. Column comparisons, defined as conditions that compare two columns using non-equality operators such as *greater than*, comprise 3% 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.

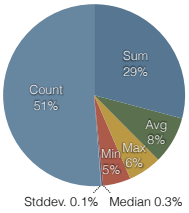
Join type. Inner join is the most common join type (69%), followed by left join (29%) and cross join (1%). The remaining types (right join and full join) together account for less than 1%.

Question 5: What fraction of queries use aggregations?



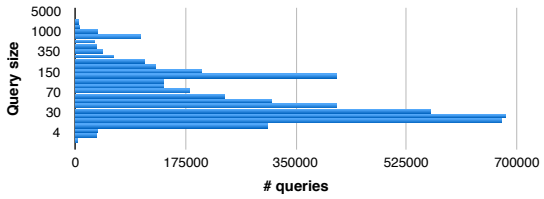
Results. Approximately one-third of queries are statistical, meaning they return only aggregations (count, average, etc.). The remaining queries return non-aggregated results (i.e., raw data) in at least one output column.

Question 6: Which aggregation functions are used most frequently?



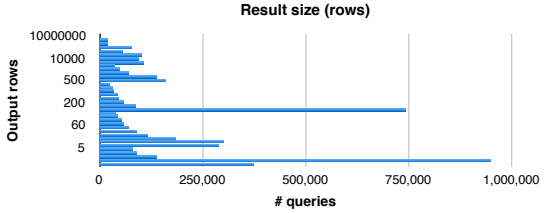
Results. `Count` is the most common aggregation function (51%), followed by `Sum` (29%), `Avg` (8%), `Max` (6%) and `Min` (5%). The remaining functions account for fewer than 1% of all aggregation functions.

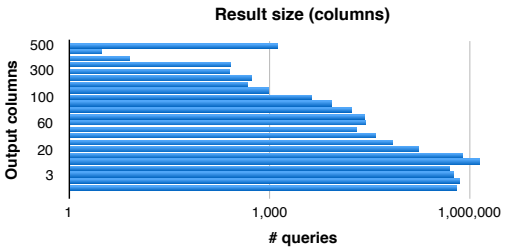
Question 7: How complex are typical queries?



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.

2.2 Summary 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 the remainder of this paper 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. This validates the focus on counting and histogram queries by the majority of previous general-purpose differential privacy mechanisms [14, 40, 42, 46]. Our work similarly focuses on this class of queries. In Section 3.7.2 we discuss possible extensions to support other aggregation functions.

2.3 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.

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

Several existing general-purpose differential privacy mechanisms support queries with join. Table 1 summarizes these mechanisms

	Requirement 1	Requirement 2		
	Database compatibility	One-to-one equijoin	One-to-many equijoin	Many-to-many equijoin
PINQ [40]		✓		
wPINQ [46]		✓	✓	✓
Restricted sensitivity [14]		✓	✓	
DJoin [42]		✓		
Elastic sensitivity (this work)	✓	✓	✓	✓

Table 1: Comparison of general-purpose differential privacy mechanisms with support for join.

and their supported features in comparison to our proposed mechanism (last row). In Section 6 we discuss additional mechanisms which are not considered here either because they are not general-purpose or they cannot support joins.

PINQ. Privacy Integrated Queries (PINQ) [40] 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.

wPINQ. Weighted PINQ (wPINQ) [46] 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, but the utility of this approach for general queries is unknown due to evaluation only on graph triangle counting queries. wPINQ additionally does not satisfy our database compatibility requirement. The system described by Proserpio et al. [46] uses a custom runtime; applying wPINQ in an existing database would require modifying the database to propagate weights during execution.

Restricted sensitivity. Restricted sensitivity [14] 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. [14] 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 [42] 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.

To address the limitations of existing mechanisms we propose *elastic sensitivity*, discussed next. Elastic sensitivity is compatible with any existing database and supports general equijoins with the full spectrum of join relationships. This combination allows use of elastic sensitivity in real-world settings.

3. 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 in Section 3.2, formally define elastic sensitivity in Section 3.3, give an example in Section 3.4, prove its correctness in Section 3.5, and discuss two optimizations in Section 3.6.

3.1 Motivation

Many previous differential privacy mechanisms [14,40] 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." [40]

Techniques based on local sensitivity [22, 44] 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 [44] 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 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 are 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 Section 3.5, 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 4, we describe how to use elastic sensitivity to enforce differential privacy.

3.2 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 [25].

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.

The formal definition of differential privacy is written in terms of the *distance* between two databases: $d(x,y) = |(x-y) \cup (y-x)|$. Two databadses $x, y \in D^n$ are *neighbors* if $d(x,y) = 1$. In this paper we consider *unbounded* differential privacy [36], in which x can be obtained from its neighbor y only by addition or removal of one tuple from y.

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:*

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

One measure of sensitivity is *global sensitivity*, which is the maximum difference between the query's result on *any* two possible databases with distance 1:

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 [40] 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* [22,44], 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. 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 picking a table $t_i \in x$ and adding or removing a single tuple from it, equivalent to adding or removing a single tuple from 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 [22, 44] 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:*

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

3.3 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 1 contains the complete definition, which is 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 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).

Our notation admits subqueries with aggregation. For example, the query "how many cities have had more than 10 trips" can be written:

$$\mathit{Count}(\sigma_{\text{count}>10}(\mathit{Count}_{\text{city\_id}}(\text{trips})))$$

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 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 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 4 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 1(c).

Ancestors of a relation. The definition in Figure 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 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. r_1 and r_2 are non-overlapping when $|\mathcal{A}(r_1) \cap \mathcal{A}(r_2)| = 0$.

Join conditions. For simplicity our notation refers only to the case where a join contains a single equality predicate. However, 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.

Unsupported queries. We discuss several cases of queries that are not supported by our definition in Section 3.7.1.

3.4 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 [46]. This example contains multiple self-joins, which demonstrate the most complex recursive cases of Figure 1.

Following Prosperio et al. we select privacy budget $\epsilon = 0.7$ and consider the `ca-HepTh` [6] 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:

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

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

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

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

$$\begin{aligned}&(65+k) + (65+k) + 1\\ =&131 + 2k\end{aligned}$$

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:

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

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 4.1, elastic sensitivity must be smoothed using smooth sensitivity [44] 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 4.1).

3.5 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 $\hat{S}_R$.

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 add k rows with 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 $r_1 \underset{a=b}{\bowtie} r_2$. We need to show that:

Core relational algebra:

Attribute names

a

Value constants

v

Relational transformations

$$\begin{array}{rcl} R & ::= & t \mid R_1 \underset{x=y}{\bowtie} R_2 \\ & \mid & \Pi_{a_1,\ldots,a_n} R \mid \sigma_\varphi R \\ & \mid & \mathit{Count}(R) \\ & \mid & \mathit{Count}_{G_1..G_n}(R) \end{array}$$

Selection predicates

$$\begin{array}{rcl} \varphi & ::= & a_1 \theta a_2 \mid a \theta v \\ \theta & ::= & < \mid \leq \mid = \\ & \mid & \neq \mid \geq \mid > \end{array}$$

Counting queries

$$\begin{array}{rcl} Q & ::= & \mathit{Count}(R) \\ & \mid & \underset{G_1..G_n}{\mathit{Count}}(R) \end{array}$$

(a)

Definition of elastic stability:

$$\begin{array}{lcl} \hat{S}^{(k)}_R & :: & R \to D^n \to \textit{elastic stability} \\ \hat{S}^{(k)}_R(t,x) & = & 1 \\ \hat{S}^{(k)}_R(r_1 \underset{a=b}{\bowtie} r_2, x) & = & \end{array}$$

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

$$\begin{array}{lcl} \hat{S}^{(k)}_R(\Pi_{a_1,\ldots,a_n} r, x) & = & \hat{S}^{(k)}_R(r,x) \\ \hat{S}^{(k)}_R(\sigma_\varphi r, x) & = & \hat{S}^{(k)}_R(r,x) \\ \hat{S}^{(k)}_R(\mathit{Count}(r)) & = & 1 \\ \hat{S}^{(k)}_R(\mathit{Count}_{G_1..G_n}(r)) & = & \hat{S}^{(k)}_R(r,x) \end{array}$$

Definition of elastic sensitivity:

$$\begin{array}{lcl} \hat{S}^{(k)} & :: & Q \to D^n \to \textit{elastic sensitivity} \\ \hat{S}^{(k)}(\mathit{Count}(r), x) & = & \hat{S}^{(k)}_R(r,x) \\ \hat{S}^{(k)}(\underset{G_1..G_n}{\mathit{Count}}(r), x) & = & \hat{S}^{(k)}_R(r,x) \end{array}$$

(b)

Maximum frequency at distance k:

$$\begin{array}{lcl} \mathsf{mf}_k & :: & a \to R \to D^n \to \mathbb{N} \\ \mathsf{mf}_k(a,t,x) & = & \mathsf{mf}(a,t,x) + k \\ \mathsf{mf}_k(a, r_1 \underset{a=b}{\bowtie} r_2, x) & = & \mathsf{mf}_k(a,r_1,x)\mathsf{mf}_k(b,r_2,x) \\ \mathsf{mf}_k(b, r_1 \underset{a=b}{\bowtie} r_2, x) & = & \mathsf{mf}_k(a, r_1 \underset{a=b}{\bowtie} r_2, x) \\ \mathsf{mf}_k(a, \Pi_{a_1,\ldots,a_n} r, x) & = & \mathsf{mf}_k(a,r,x) \\ \mathsf{mf}_k(a, \sigma_\varphi r, x) & = & \mathsf{mf}_k(a,r,x) \end{array}$$

(c)

Ancestors of a relation:

$$\begin{array}{lcl} \mathcal{A} & :: & R \to \{R\} \\ \mathcal{A}(t) & = & \{t\} \\ \mathcal{A}(r_1 \underset{a=b}{\bowtie} r_2) & = & \mathcal{A}(r_1) \cup \mathcal{A}(r_2) \\ \mathcal{A}(\Pi_{a_1,\ldots,a_n} r) & = & \mathcal{A}(r) \\ \mathcal{A}(\sigma_\varphi r) & = & \mathcal{A}(r) \end{array}$$

(d)

Figure 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.

$$\mathsf{mf}_k(a, r_1 \underset{a=b}{\bowtie} r_2, x) \geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a, r_1 \underset{a=b}{\bowtie} r_2, y) \quad (1)$$

In the worst case, each row from r_1 matches each row from r_2 (i.e. a cartesian product). So we can rewrite equation 1:

$$\mathsf{mf}_k(a, r_1 \underset{a=b}{\bowtie} r_2, x) \geq \max_{y:d(x,y)\leq k} \mathsf{mf}(a,r_1,y)\mathsf{mf}(b,r_2,y) \quad (2)$$

Then, we rewrite the left-hand size to be equal to the right, 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:

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

Which is equal to the right-hand side of equation 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 added or removed 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 $\hat{S}^{(k)}_R(r_2,x)$ new rows, producing at most $\mathsf{mf}_k(a,r_1,x)\hat{S}^{(k)}_R(r_2,x)$ new 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 $\mathsf{mf}_k(b,r_2,x)\hat{S}^{(k)}_R(r_1,x)$ new rows.

We choose to modify the relation resulting in the largest number of new 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 adding a single row to the database could result in new rows in *both* r_1 and r_2.

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

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

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

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

joined relation contains $\hat{S}^{(k)}_R(r_1, x)\hat{S}^{(k)}_R(r_2, x)$ new rows. The total number of new rows is therefore bounded by the sum of the bounds on the three classes:

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

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.

Case *Count*$_{G_1..G_n}(r)$. The relational *Count* with grouping produces a relation representing a histogram, with one count per group. Each additional row in the underlying relation can result in at most one new or modified group. By the inductive hypothesis, the total number of new rows is bounded by $\hat{S}^{(k)}_R(r, x)$. □

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 $A^{(k)}_q(x)$ of q executed at distance k from database x:*

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

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). By McSherry [40] Theorem 4 (parallel composition), the sensitivity of a histogram query is equal to the maximum of the individual sensitivities for each group. In the worst case, all of the rows in r end up in the same group, and the sensitivity of the query is equal to the stability of r. Once again, the result follows from Lemma 2. □

3.6 Using Data Models for Tighter Bounds on Local Sensitivity

Local sensitivity is defined over neighboring databases of the true database. Our definition of elastic sensitivity considers all neighboring databases to be valid. However, certain neighboring databases are infeasible and therefore need not be considered: in our dataset, for example, city data is publicly known and does not change, and user IDs are never duplicated for multiple users. Both kinds of information can be used to prune the set of neighboring databases considered, and thereby tighten our bound on local sensitivity, without compromising our privacy guarantees.

Public tables. For joins on public tables, our implementation of elastic sensitivity considers only the effects on the private table when calculating sensitivity. For example, because city data is publicly available, joining trips with cities poses no additional privacy threat compared to querying trips directly.

More precisely, in a join expression `T1 JOIN T2 ON T1.A = T2.B`, if `T2` is publicly known, the sensitivity of the join is the maximum frequency of `T2.B` (the frequency of the join key in `T1` is ignored). Note this formulation correctly prevents the use of a publicly-known table with repeated join keys from revealing information about a private table.

Unique join keys in private tables. Many private tables have primary key columns that will never contain duplicates, making many neighboring databases impossible. For example, in the *users* table, each user corresponds to a unique ID. In the *trips* table, however, a user ID may appear multiple times—once for each trip.

A naive (but correct) sensitivity bound for a relation joining these two tables considers a neighboring database in which the *users* table contains a duplicate user, resulting in duplication of each of that user's trips in the output of the join. Our data model optimization recognizes that this neighboring database violates data integrity conditions and therefore cannot occur.

More precisely, in a join expression `R1 JOIN R2 ON R1.A = R2.B`, if the data model asserts that `R1.A` is a unique key, then the sensitivity of the join is equal to the sensitivity of `R2`.

The set of public tables is domain-specific and will vary in each data environment. The unique key columns can be obtained directly from the database schema.

3.7 Discussion of Limitations and Extensions

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

3.7.1 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 our empirical study, 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.

3.7.2 Supporting Other Aggregation Functions

In this section we outline possible 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 $\mathsf{vr}(a, r)S_R^{(k)}(r, x)$. 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 [22], and therefore *any* differential privacy technique will provide poor utility in the general case.

Due to this fundamental limitation, previous work [22, 44, 49] 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.

4. FLEX: PRACTICAL DIFFERENTIAL PRIVACY FOR SQL QUERIES

This section describes FLEX, our system to enforce differential privacy for SQL queries using elastic sensitivity. Figure 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. FLEX then applies smooth sensitivity [44] to the elastic sensitivity and finally adds noise drawn from the Laplace distribution to the original query results. In Section 4.1 we prove this approach provides (ϵ, δ)-differential privacy.

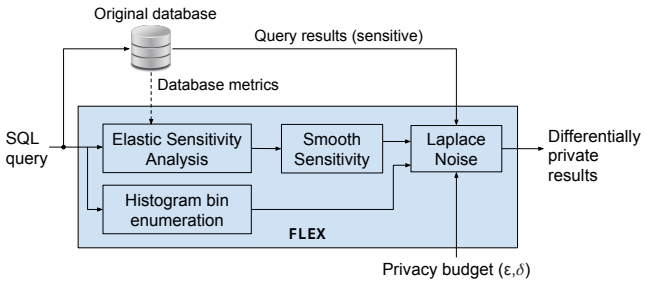


Figure 2: Architecture of FLEX

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 interactions with the database to enforce differential privacy. As we demonstrate in Section 5, 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 2 is ideal for environments where database updates are far less frequent than database queries.

Most databases can be configured using triggers [3] 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. We evaluate the runtime and success rate of this analysis in Section 5.

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 3.3 assumes that the analyst provides the desired histogram bins labels ℓ. This requirement, also adopted by previous work [40], 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 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.

This process 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.

4.1 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. [44]:

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 $q_\ell(x) + Lap(2S/\epsilon)$.*

This mechanism leverages smooth sensitivity [44], 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. [44] 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. □

4.2 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 [23] 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* [26] improves on this method to produce a tighter bound on the privacy budget used. Both approaches are independent of the mechanism and thus 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* [24] answers only queries whose results lie above a predefined threshold. This approach depletes the privacy budget for answered queries only. The *Matrix Mechanism* [37] and *MWEM* [30] 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. [19] 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.

The *Exponential Mechanism* [39] 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.

	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 2: Performance of FLEX-based differential privacy

5. EXPERIMENTAL EVALUATION

We evaluate our approach with the following experiments:

- We measure the performance overhead and success rate of FLEX on real-world queries (Section 5.1).
- We investigate the utility of FLEX-based differential privacy for real-world queries with and without joins (Section 5.2).
- We evaluate the effect of the privacy budget ϵ on the utility of FLEX-based differential privacy (Section 5.3).
- We measure the utility impact of the data model optimizations described in Section 3.6 (Section 5.4).
- We compare FLEX and wPINQ on a set of representative counting queries using join (Section 5.5).

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 9862 real queries executed during October 2016. To build this dataset, we identified all counting queries (including histogram queries) submitted during this time which examined sensitive trip data. Our dataset also includes original results for each of these queries.

5.1 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 3.7.1. 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 2 compared with the original query execution time. We report the results in Table 2. Parsing and analysis of the query to calculate elastic sensitivity took an average of 7.03

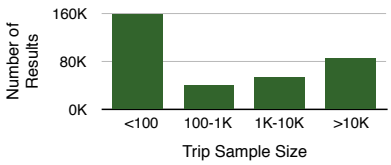


Figure 3: Distribution of population sizes for queries in our dataset

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%.

5.2 Utility of FLEX on Real-World Queries

Our work is the first to evaluate differential privacy on a set of real-world queries executed on real data. In contrast with previous evaluations of differential privacy [15, 32, 33], 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.

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 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 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 [22].

Figure 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 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 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 data model optimizations described in Section 3.6. 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%.

Query	Description	# Joins
Q1	Billed, shipped, and returned business	0
Q4	Priority system status and customer satisfaction	0
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: Evaluated TPC-H queries.

Hay et al. [32] 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.

5.2.1 Utility of FLEX on TPC-H benchmark

We repeat our utility experiment using TPC-H [17], 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 [17]. This experiment measures the ability of our system to handle complex queries and provide high utility in a new domain.

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. 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 $\delta = n^{-\epsilon \ln n}$. We marked all 8 tables as protected (non-public) so the analysis did not use the public table optimization for any of the queries.

Results. The results are shown in Figure 5. The results show the same trend as the previous experiment: error decreases with increasing population size, and this trend is observed for queries with joins and without joins. Elastic sensitivity produces high utility (less than 10% error) for all of the evaluated queries.

5.2.2 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 [25] , "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 sam-

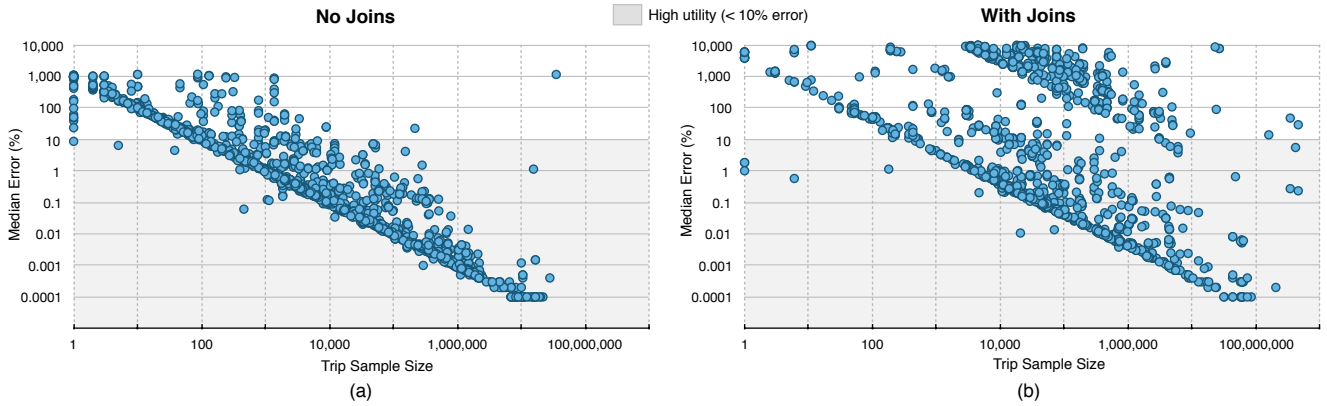


Figure 4: Median error vs population size for queries with no joins (a) and with joins (b).

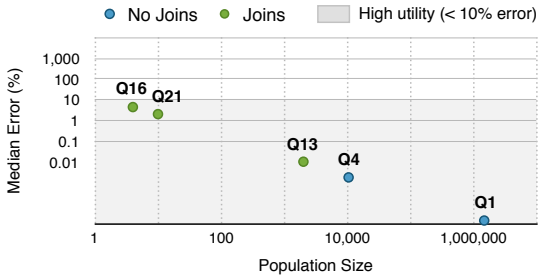


Figure 5: Median error vs population size for queries from TPC-H benchmark.

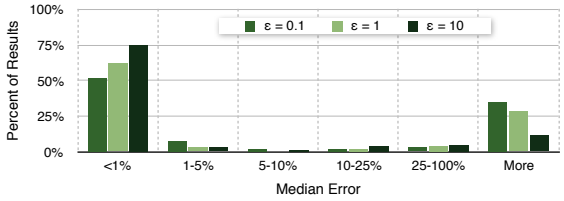


Figure 6: Effect of ϵ on median error for FLEX-based differential privacy.

ple 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.

5.3 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 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 fully half 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 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.

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

Figure 7: Manual categorization of queries with high error.

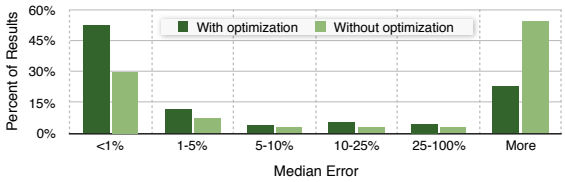


Figure 8: Impact of data model optimization for tighter bounds on sensitivity.

We summarize the results in Figure 7. 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 described in Section 3.6. 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.

5.4 Impact of Data Model Optimization

Section 3.6 describes an optimization for elastic sensitivity that

uses data models to achieve a tighter bound on sensitivity. 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 50.3% of queries in our dataset. Over half (52.3%) of these queries join private tables on unique keys; the remaining queries join private tables with public tables. All queries in the latter category benefit from the unique key optimization. This is due to the two public tables in our environment (cities and promotions) being used exclusively in joins on unique keys (e.g., city id).

Figure 8 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 30% to 52%. 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.

5.5 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 Section 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, 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 elastic sensitivity's public table optimization.

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 [13].

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 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 73% lower for program 5. FLEX and wPINQ provide identical error for program 3 due to equivalent effects of elastic sensitivity's public table optimization and wPINQ's `select` operator.

FLEX provides higher error for program 5 although 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. [46] 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. However, the authors achieve this improved utility via an alternative query carefully crafted for the MCMC process. As the behavior of MCMC is highly dependent on the alternate query, and the authors do not describe how to construct this query for general analytics programs, we are unable to evaluate wPINQ with this additional step.

6. RELATED WORK

Differential privacy was originally proposed by Dwork [20, 21, 23], and the reference by Dwork and Roth [25] 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 Section 2.4.

Lu et al. [38] 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 [47] 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 [28, 29] enforces differential privacy for functional programs, but does not support one-to-many or many-to-many joins.

Propose-test-release [22] (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 [44] 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 [41] is a practical system that leverages the sample & aggregate framework to enforce differential privacy for general-purpose analytics.

A number of less-general mechanisms for performing specific graph analysis tasks have been proposed [31, 34, 35, 48]. 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 [16] supports general equijoins in the context of graph analyses, but is restricted

Program	Joined tables	Median Population Size	Median Error (%) wPINQ	Median Error (%) 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	1.3
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: Count daily trips by city (for all cities) on Oct. 24, 2016.	trips, cities	87	11.5	11.5
5. Histogram: Count total trips per driver in Hong Kong between Sept. 9 and Oct. 3, 2016.	trips, drivers	1	974	1501
6. Histogram: Count 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	13.9

Table 4: Utility comparison of wPINQ and FLEX for selected set of representative counting queries using join.

to monotonic queries and in the worst case, runs in time exponential in the number of participants in the database.

Kifer et al. [36] 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.

7. CONCLUSION

This paper takes a first step towards practical differential privacy for general-purpose SQL queries. To meet the requirements of real-world SQL queries, we proposed elastic sensitivity, the first efficiently-computed approximation of local sensitivity that supports joins. We have released an open-source tool for computing elastic sensitivity of SQL queries [4]. We use elastic sensitivity to build FLEX, a system for enforcing differential privacy for SQL queries. We evaluated FLEX on a wide variety of queries, demonstrating that FLEX can support real-world queries and provides high utility on a majority of queries with large population sizes.

Acknowledgments

The authors would like to thank Abhradeep Guha Thakurta, Om Thakkar, and Arjun Baokar for their helpful comments on an earlier version of this paper. This material is in part based upon work supported by DARPA contract #N66001-15-C-4066 and by the Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

8. REFERENCES

[1] Apple previews iOS 10, the biggest iOS release ever. http://www.apple.com/newsroom/2016/06/apple-previews-ios-10-biggest-ios-release-ever.html.

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

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

[4] Dataflow analysis & differential privacy for SQL queries. http://www.github.com/uber/sql-differential-privacy.

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

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

[7] Morgan Stanley Breach a Reminder of Insider Risks. https://securityintelligence.com/news/morgan-stanley-breach-reminder-insider-risks/.

[8] Nearly 5,000 patients affected by UC Irvine medical data breach. http://www.latimes.com/business/la-fi-uc-irvine-data-breach-20150618-story.html.

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

[10] Sutter Health California Pacific Medical Center audit uncovers data breach. http://www.csoonline.com/article/2876324/data-breach/sutter-health-california-pacific-medical-center-audit-uncovers-data-breach.html.

[11] Swiss spy agency warns U.S., Britain about huge data leak. http://www.reuters.com/article/us-usa-switzerland-datatheft-idUSBRE8B30ID20121204.

[12] Uber Releases Open Source Project for Differential Privacy. https://medium.com/uber-security-privacy/differential-privacy-open-source-7892c82c42b6.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

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