2020 IEEE European Symposium on Security and Privacy (EuroS&P)

CHORUS: a Programming Framework for Building Scalable Differential Privacy Mechanisms

Noah Johnson
UC Berkeley
noahj@berkeley.edu

Joseph P. Near
University of Vermont
jnear@uvm.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Dawn Song
UC Berkeley
dawnsong@berkeley.edu

***Abstract*—Differential privacy is fast becoming the gold standard in enabling statistical analysis of data while protecting the privacy of individuals. However, practical use of differential privacy still lags behind research progress because research prototypes cannot satisfy the scalability requirements of production deployments. To address this challenge, we present CHORUS, a framework for building scalable differential privacy mechanisms which is based on cooperation between the mechanism itself and a high-performance production database management system (DBMS). We demonstrate the use of CHORUS to build the first highly scalable implementations of complex mechanisms like Weighted PINQ, MWEM, and the matrix mechanism. We report on our experience deploying CHORUS at Uber, and evaluate its scalability on real-world queries.**

***Index Terms*—privacy, differential privacy, SQL queries, query rewriting, security**

1. Introduction

Organizations are collecting more and more sensitive information about individuals. As this data is highly valuable for a broad range of business interests, organizations are motivated to provide analysts with flexible access to the data. At the same time, the public is increasingly concerned about privacy protection. There is a growing and urgent need for technology solutions that balance these interests by supporting general-purpose analytics while guaranteeing privacy protection.

Differential privacy [28], [32] is widely recognized by experts as the most rigorous theoretical solution to this problem. Differential privacy provides a formal guarantee of privacy for individuals while allowing general statistical analysis of the data. In short, it states that the presence or absence of any single individual's data should not have a large effect on the results of a query. This allows precise answers to questions about populations in the data while guaranteeing the results reveal little about any individual. Unlike alternative approaches such as anonymization and k-anonymity, differential privacy protects against a wide range of attacks, including attacks using auxiliary information [27], [61], [65], [77].

Current research on differential privacy focuses on development of new algorithms, called *mechanisms*, to achieve differential privacy for a particular class of queries. Researchers have developed dozens of mechanisms covering a broad range of use cases, from general-purpose statistical queries [19], [30], [54], [55], [59], [64], [67] to special-purpose analytics tasks such as graph analysis [24], [42], [46], [47], [72], linear queries [8], [25], [41], [50]–[53], [68], [81], [82], [85], and analysis of data streams [31], [74].

Despite extensive academic research and an abundant supply of mechanisms, differential privacy has not been widely adopted in practice. Existing applications of differential privacy in practice are limited to specialized use cases [2], [34].

A major challenge for the practical adoption of differential privacy is the ability to deploy differential privacy mechanisms *at scale*. Today's data analytics infrastructures include industrial-grade database management systems (DBMSs) carefully tuned for performance and reliability, designed to process datasets consisting of billions of rows.

The simplest mechanisms for differential privacy, like the *Laplace mechanism* [30], answer an analyst's query by adding noise to the final result of the query. This mechanism can be easily deployed atop an existing high-performance DBMS by leveraging the DBMS to execute the analyst's query, then adding the right amount of noise to the result. Since it uses the DBMS to perform the actual data processing tasks, this approach scales well. Some existing work, such as FLEX [44], takes this approach, which we call the *post-processing* architecture. For appropriate mechanisms, the post-processing architecture solves the scalability problem.

However, more advanced differential privacy mechanisms require fundamental changes to the way queries execute. Summation queries, for example, require *clipping* the data before summing it to control the influence of outliers. The post-processing approach is fundamentally incompatible with mechanisms like this one—it is *impossible* to run the analyst's query unmodified and then achieve differential privacy by post-processing the results. Unfortunately, the vast majority of recently-developed mechanisms fall into this category (e.g. [8], [24], [25], [41], [42], [46], [47], [50]–[53], [68], [72], [81], [82], [85]), and cannot be implemented using the post-processing architecture.

As a result, no scalable implementation exists for many of the exciting differential privacy mechanisms developed in recent years. The implementations which have been developed (e.g. [49], [55], [59], [67], [80]) modify or replace the DBMS with a custom engine, which is unlikely to offer performance on par with modern production DBMSs.

The CHORUS Framework. This paper describes CHO-

DOI 10.1109/EuroSP48549.2020.00041

RUS, a framework for developing and deploying cutting-edge differential privacy mechanisms at scale. CHORUS makes it easy to develop mechanism implementations which work *in cooperation with* an existing high-performance DBMS, even for mechanisms which require modifying queries or generating entirely new ones. CHORUS supports scalable implementations by leveraging the DBMS for data processing tasks, instead of custom code. We call this the *cooperative* architecture for differential privacy mechanisms.

CHORUS provides a programming framework to support implementing mechanisms in the cooperative architecture. The framework has three major components: **rewriting**, for modifying queries to perform functions like clipping; **analysis**, for analyzing queries to determine properties like how much noise is required for differential privacy; and **post-processing**, for processing the results of executing queries. To implement a summation mechanism with clipping, for example, we can use CHORUS's rewriting component to modify the analyst's query so that *the DBMS performs the clipping* as well as the summation. With this modification, the rest of the mechanism can be implemented via analysis and post-processing of the rewritten query.

CHORUS supports integration with *any* standard SQL database. The framework is designed to facilitate working directly with SQL queries, since SQL is the most commonly used language for high-performance production DBMSs. By using a standard SQL database engine instead of a custom runtime, CHORUS can leverage the reliability, scalability and performance of modern databases, which are built on decades of research and engineering experience.

The cooperative architecture applies to *all* of the recently-developed differential privacy mechanisms—even ones that require significant changes to the way queries execute or generate entirely new queries. We demonstrate the flexibility of the approach, and of the CHORUS framework, by implementing both simple mechanisms like summation with clipping and complex mechanisms like wPINQ, MWEM, and the matrix mechanism. In all of these implementations, CHORUS supports scalability by moving data processing tasks to the DBMS.

Deployment. We have made CHORUS available as open source software [3], and it is designed for integration in production environments. We describe how to deploy CHORUS to provide differential privacy in the face of untrusted analysts who may submit malicious queries, and present practical strategies for privacy budget management as part of a CHORUS deployment.

We report on our experience deploying CHORUS at Uber for its internal analytics tasks. CHORUS represents a significant part of the company's General Data Protection Regulation (GDPR) [4] compliance efforts, and provides both differential privacy and access control enforcement.

Evaluation. We evaluate the scalability of CHORUS on real queries written by analysts at Uber, using a 300 million rows sampled from the production data. Our evaluation results demonstrate that mechanism implementations built with CHORUS are capable of scaling to real-world analysis tasks.

Contributions. In summary, we make the following contributions:

1) We present the CHORUS framework, which enables a novel *cooperative architecture* for implementing differential privacy mechanisms atop high-performance DBMSs (§ 3).
2) We demonstrate CHORUS's flexibility by developing scalable implementations for a number of advanced differential privacy mechanisms (§ 5).
3) We release CHORUS as open source [3] and describe how to deploy it to provide differential privacy in production settings (§ 6).
4) We report on our experience deploying CHORUS to enforce differential privacy at Uber, where it processes more than 10,000 queries per day (§ 6).
5) We demonstrate the scalability of CHORUS by evaluating it on 18,774 real-world queries with a database of 300 million rows (§ 7).

2. Background

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

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

Differential privacy can be enforced by adding noise to the non-private results of a query. The scale of this noise depends on the *sensitivity* of the query. The *global sensitivity* of a query $f : D^n \to \mathbb{R}$ is defined as:

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

Importantly, differential privacy mechanisms satisfy a *sequential composition* property: if F_1 satisfies (ϵ_1, δ_1)-differential privacy, and F_2 satisfies (ϵ_2, δ_2)-differential privacy, then running both F_1 and F_2 satisfies $(\epsilon_1+\epsilon_2, \delta_1+\delta_2)$-differential privacy. For more on differential privacy, see Dwork and Roth [32].

Statistical queries. Differential privacy aims to protect the privacy of individuals in the context of *statistical queries*. In SQL terms, these are queries using standard aggregation operators (`COUNT`, `AVG`, etc.) as well as histograms created via the `GROUP BY` operator in which aggregations are applied to records within each group. Differential privacy is not suitable for queries that return raw data (e.g. rows in the database) since such queries are inherently privacy-violating.

Mechanism design. Research on differential privacy has produced a large and growing number of differential privacy mechanisms. Some mechanisms are designed to provide broad support for many types of queries [19], [30], [49], [54], [55], [59], [64], [67], while others are designed to produce maximal utility for a particular application [8], [24], [25], [34], [41]–[43], [46], [47], [50], [52], [68], [72],

[81], [82], [85]. While mechanisms adopt unique strategies for enforcing differential privacy in their target domain, they generally share a common set of design choices and algorithmic components. For example, many mechanisms require addition of Laplace noise to the result of the query.

3. The CHORUS Architecture

This section presents the system architecture and advantages of CHORUS, and compares it against existing architectures for differentially private analytics. We first describe the design goals motivating the CHORUS architecture. Then, in Section 3.1, we describe the limitations of existing architectures preventing previous work from attaining these goals. Finally, Section 3.2 describes the novel architecture of CHORUS and provides an overview of our approach.

Design Goals. The design of CHORUS is motivated by the desire to enforce differential privacy at the scale of real-world industrial deployments. To that end, CHORUS has the following design goals:

- Process data using a DBMS, not a custom system
- Support a broad range of privacy mechanisms
- Integrate easily with existing data environments

As we will demonstrate in the next section, achieving these goals is challenging, and no existing system manages to achieve all three. We emphasize the importance of integration with an existing, highly-tuned database management system (DBMS)—such systems are the result of decades of research, and the massive scale of modern data warehouses is made possible only by leveraging these results. A custom-built system specific to differential privacy is unlikely to ever match the performance of a highly-tuned DBMS designed for big data.

Motivating example: bounded sum queries. Consider a simple example query over a table called trips containing information about taxi trips. Suppose we want to return the sum of miles driven over all of the trips in the database. We might use a query like this:

```
SELECT SUM(trip_distance) FROM trips
```

Satisfying differential privacy for this query is challenging, because there is no obvious bound on the global sensitivity of the SUM. Two neighboring databases differ in only a single row, but that row may have any *value*, and adding a row to the database increases the sum by the *value* of an attribute in the new row. Without some upper bound on the attribute values rows can have, it is not possible to bound the sensitivity of the summation query.

The usual strategy for solving this problem is *clipping*: we first *enforce* a bound on the maximum distance of any trip in the database, then perform the sum on the clipped distances. We can implement this strategy using a revised query:

```
SELECT SUM(max(0, min(100, trip_distance)))
FROM trips
```

The revised query has a global sensitivity of 100, because all trip distances are clipped to lie between 0 and 100 miles. We can achieve differential privacy for the revised query by adding Laplace noise scaled to $\frac{100}{\epsilon}$ [32].

3.1. Existing Architectures

Existing systems for enforcing differential privacy for data analytics tasks adopt one of two architecture types: they are either *deeply integrated* systems or *post processing* systems. These architectures are summarized in Figure 1(a) and Figure 1(b). PINQ [55], Weighted PINQ [67], GUPT [59], and Airavat [71] follow the *deep integration* architecture: each one provides its own specialized DBMS, and cannot be used with a standard DBMS. As described earlier, the use of a specialized DBMS is likely to prevent the use of these systems for large-scale deployments.

The FLEX [44] system uses the *post processing* architecture: it runs the analyst's original query on the database, then adds noise to the result. This approach supports mechanisms that do not modify the semantics of the original query, like Elastic Sensitivity, PINQ, and Restricted sensitivity. The major advantage of the post processing architecture is that it is compatibile with existing DBMSs.

However, the post processing architecture is *fundamentally incompatible* with mechanisms that change how the original query executes—such as queries that perform clipping, like the motivating example above. FLEX is not capable of answering SUM queries with differential privacy unless bounds on the values of the summed columns are known *a priori* (which is often not the case).

Many differential privacy mechanisms make even more complicated changes to the query the analyst actually wants to answer. For example, Sample & Aggregate splits the database into chunks and runs the analyst's query on each chunk separately, then aggregates the results, and WPINQ assigns a weight to each row of the database and updates these weights as the query executes. More recent algorithms, like MWEM, the Matrix Mechanism, and others are even more complicated. These approaches are impossible to implement via post-processing: they require running queries which are *different* from the analyst's original query, and they often require multi-stage *interaction* with the DBMS.

The *deeply integrated* and *post processing* architectures in Figure 1(a) and (b) therefore both fail to address two major challenges in implementing a practical system for differentially private data analytics:

- Deeply integrated systems use custom DBMSs, which are unlikely to achieve parity with mature DBMSs in terms of performance and scalability, query optimization, recoverability, and distribution.
- Neither architecture supports all of the different mechanisms discussed earlier. The deeply integrated architecture requires building a new DBMS for each mechanism, while the post processing architecture is inherently incompatible with some mechanisms.

3.2. The CHORUS Architecture

In CHORUS, we propose a novel alternative, which we call the *cooperative* architecture. As shown in Figure 1(c), the cooperative architecture has two major differences with existing architectures:

- CHORUS integrates tightly with an existing unmodified SQL DBMS, which holds the sensitive data.

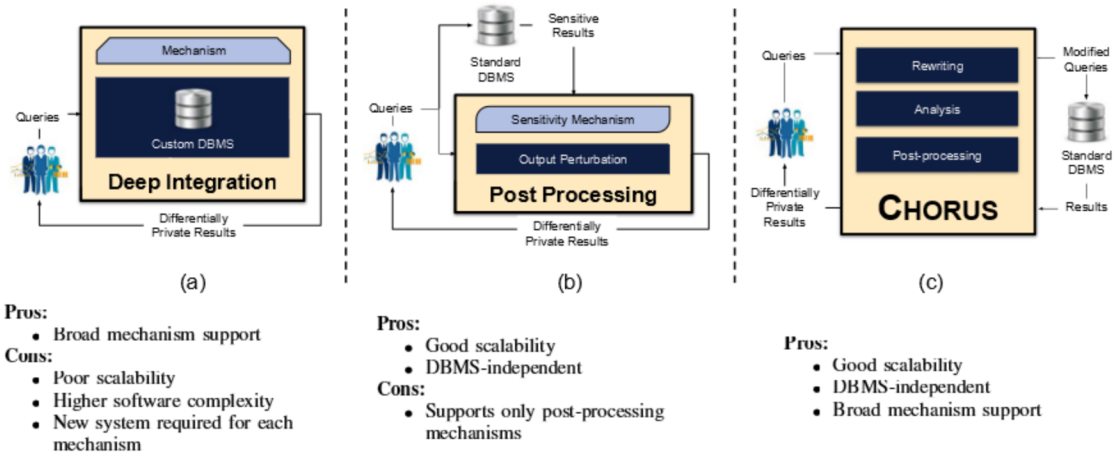

Figure 1: Existing architectures: (a), (b); architecture of CHORUS: (c).

- CHORUS can perform post-processing operations *and* modify the way queries execute, enabling implementations of all the mechanisms discussed earlier.

CHORUS provides a flexible framework for implementing differential privacy mechanisms in the cooperative architecture. In this architecture, the analyst specifies the task to be completed via one or more queries to be answered, plus additional metadata (for example, the desired values for the privacy parameter(s)). CHORUS provides the following components:

- **Rewriting** supports generating new SQL queries, by modifying queries from the analyst's workload or generating new ones.
- **Analysis** supports analyzing queries in the workload to determine their properties (e.g. sensitivity).
- **Post-processing** supports post-processing of query results (e.g. to combine results or add noise).

For counting queries like those supported by FLEX [44], CHORUS can simply execute the queries in the workload and add noise to the results via post-processing. For a SUM query requiring clipping, like our motivating example, CHORUS rewrites the query to include the clipping bound specified by the analyst in the metadata, executes the rewritten query on the DBMS, and adds noise to the result. For more complicated mechanisms like the Matrix Mechanism [51], CHORUS may generate entirely new queries not present in the original workload. CHORUS has two key advantages over previous work:

- CHORUS is DBMS-independent (unlike the deeply integrated approach): it requires neither modifying the database nor switching to purpose-built database engines. Our approach can therefore leverage existing high-performance DBMSs to scale to big data.
- CHORUS can implement a wide variety of privacy-preserving techniques. Unlike the post processing approach, CHORUS is compatible with all of the mechanisms discussed earlier, and many more.

CHORUS's architecture is specifically designed to be easily integrated into existing data environments. We report on the deployment of CHORUS at Uber in Section 6.

Challenges. Developing differential privacy mechanisms to target the cooperative architecture is much more challenging than developing either deeply integrated or post-processing solutions. In particular, this model requires interacting closely with a DBMS which may have a non-standard dialect or feature set, analyzing SQL queries which may have complicated structure or target a specific dialect, and generating new queries which target the appropriate DBMS.

CHORUS is designed specifically to address these challenges. We detail the solutions to each one in Section 4. We also develop a number of case studies (in Section 5) to demonstrate how the features of CHORUS support programmers in developing differential privacy mechanisms in the cooperative architecture, following the *rewrite-analyze-postprocess* structure outlined above.

3.3. Threat Model

Typical deployments of CHORUS mechanisms involve three kinds of parties: *data subjects*, who contribute sensitive data to the database, the *data curator*, who manages the database containing the sensitive data, and *analysts*, who submit queries to be answered using the sensitive data. CHORUS is designed primarily to protect the sensitive data contributed by data subjects against malicious analysts.

As with most systems designed around the central model of differential privacy, we assume that the data curator behaves honestly. The DBMS, CHORUS-based mechanisms, and other systems maintained by the data curator for answering queries are assumed to be trusted, and cannot be corrupted.

The adversary in this setting is represented by a group of one or more malicious analysts, who would like to discover a fact about an individual data subject in the sensitive data. The analyst may submit arbitrary queries to the system, designed to expose private information about an individual. These queries may be adaptively chosen based on previous results, and more than one analyst may collude to infer private information. CHORUS is designed to guarantee that each query response satisfies differential privacy, which implies a bound on the total privacy cost of all queries posed by the analyst.

Our deployment enforces that the adversary may access the sensitive data *only* via the centralized query interface. All components of the system, including the query interface, the privacy budget accountant, CHORUS, and the DBMSs themselves, are protected from tampering by the adversary via access-control protections.

3.4. Selecting a Mechanism

As described in Section 2, many differential privacy mechanisms exist, and many of these require the analyst to re-phrase queries in a new way or provide additional inputs. As a result, it is not always possible to *automatically* select the best mechanism for answering a SQL query posed by the analyst—the best mechanism might depend on domain knowledge, other queries in the workload, or the ability of the analyst to re-phrase the query. For certain classes of queries—for example, linear queries over a single database table—Hay et al. [48] demonstrate that a machine learning-based approach can leverage properties of the data to select a mechanism most likely to yield high utility.

The interface provided to the analyst in each deployment of CHORUS will therefore depend on the analyst's expected expertise in differential privacy.

- *Non-experts* will submit standard SQL queries, without any knowledge about how differential privacy is being enforced. CHORUS can attempt to select a mechanism which supports the features used in the query. If no suitable mechanism is found, CHORUS rejects the query.
- *Privacy-conscious* analysts will submit workloads of queries written in subsets of SQL (e.g. linear queries), and CHORUS can select the best approach (e.g. using a machine learning model).
- *Privacy experts* will manually select a mechanism, and phrase their queries appropriately for that mechanism (e.g. for the Sparse Vector Technique, a sequence of queries and a threshold).

The CHORUS API is designed to facilitate all three possibilities. For our prototype deployment (discussed in Section 6), we expected users to be non-experts, and implemented a simple rule-based mechanism to select a mechanism based on the aggregation function used and whether or not the query contained joins. Section 4.4 describes the use of the CHORUS API for this purpose.

3.5. Privacy Budget Management

We have designed CHORUS to be flexible in its handling of the privacy budget, since the best approach in a given setting is likely to depend on the domain and the kinds of queries posed.

One approach to budget management involves tracking a single global budget, subtracting from the budget when each query runs using standard composition. One straightforward optimization is to use *advanced composition* [33], which improves the total budget for k queries to be proportional to $\sqrt{k}$. Recent advances in composition, like Rényi differential privacy [58], zero-concentrated differential privacy [22], and truncated concentrated differential privacy [21], can be directly applied in a similar way.

Some mechanisms build a differentially private synthetic representation of the data, and use this representation to answer queries (e.g. MWEM and the matrix mechanism). This is another form of budget management: once the representation is built, it can be used to answer an unbounded number of queries without incurring additional privacy cost. Such mechanisms often offer better accuracy over workloads of queries than any composition approach which supports online answering of queries.

We describe the CHORUS API for implementing budgeting strategies in Section 4.4. Mechanism definitions return the privacy cost of one execution, and approaches to budgeting can be built on top of this interface.

3.6. Assumptions & Limitations

CHORUS's guarantees rely on the soundness of several underlying components. Bugs in these components could cause a failure of the guarantee, and the release of sensitive information without differential privacy.

Correctness of Underlying Libraries. CHORUS uses the Apache Calcite [1] framework for parsing SQL queries and translating them to a bag-based variant of relational algebra. A bug in Calcite could cause a query to be wrongly parsed or converted. Such a failure would result in incorrect results being returned to the analyst, but would not cause a failure in differential privacy, since CHORUS analyzes, rewrites, and executes the *final* output of Calcite's processing pipeline. In addition, Apache Calcite is widely used and therefore likely to be reliable.

Soundness of Abstract Interpretation. To analyze a query's sensitivity, CHORUS performs abstract interpretation of the query (see Section 5). CHORUS provides an abstract interpretation framework, which enables implementing many different kinds of analyses. A bug in either the framework or the implementation of a specific analysis could cause unsoundness in the analysis results, leading to a failure to ensure differential privacy. To mitigate this risk, we designed the framework to be compact and easily audited; further mitigation using formal verification techniques might also be possible in future work.

Semantics of DBMSs. A more subtle failure of the abstract interpretation occurs when the concrete semantics of the DBMS used to execute queries do not match the semantics encoded by the abstract interpreter. This is a special concern for CHORUS, since we aim to support many different DBMSs, and because DBMSs sometimes differ significantly in their semantics [37]. Mitigation of this risk would require formal analysis of specific DBMS implementations to verify their compliance with a standard semantics.

Dialects and Other Languages. CHORUS uses Apache Calcite [1] to parse and process queries, and works for queries that Calcite supports. Calcite does offer support for a number of SQL dialects, but it may not support all of the vendor-specific extensions offered by a particular DBMS, and CHORUS will therefore not support them either (without modifications to Calcite). Code written in languages other than SQL (including stored or user-defined functions) are not supported for the same reason.

Sources of Randomness. CHORUS relies on randomness generated by both the DBMS and the Java runtime, and

previous work has shown [57] that inadequate sources of randomness can lead to a failure of differential privacy. However, the same work also demonstrated simple solutions for improving faulty sources of randomness to recover differential privacy. Large-scale deployments of CHORUS should verify that both the DBMS and Java runtime used provide high-quality sources of randomness, or implement the appropriate countermeasures.

Correctness of Mechanism Implementations. Bugs may also exist in mechanism implementations themselves. The CHORUS framework is designed to simplify mechanism implementations to reduce bugs, but it does not eliminate them entirely. Integrating an approach for formal verification of mechanism correctness (e.g. [10], [12]–[16], [26], [35], [62], [69], [73], [79], [83], [84]) could ensure bug-free mechanisms, and is an exciting area for future work.

4. The CHORUS Programming Framework

The CHORUS framework provides a Scala library for implementing differential privacy mechanisms in the co-operative model we have proposed. The library provides support for all three components of the model:

- A *rewriting* component, with support for modifying existing SQL queries and generating new ones
- An *analysis* component, which provides an abstract interpretation framework for analyzing SQL queries
- A *post-processing* component, which provides utilities for post-processing results

In this section, we describe each of the three components. We begin with our simple motivating example from earlier: a SUM query over a column with no known upper bound. As described earlier, bounding the sensitivity of this query requires clipping the values being summed, but this is impossible to accomplish by post-processing alone.

We will implement a CHORUS mechanism to answer such queries in three steps: (1) rewrite SUM aggregations in the analyst's query to perform clipping; (2) analyze the rewritten query to determine its sensitivity; (3) run the rewritten query and add noise to the result based on the sensitivity computed in step (2).

4.1. Rewriting

For the first step, CHORUS provides a powerful *rewriting* API for modifying the analyst's queries. The following Scala code implements the rewriter for step (1) above.

```
def rewriteClip(l: Double, u: Double,
                root: Relation): Relation = {
  root.rewriteRecursive(UnitDomain) {
    (node, orig, _) =>
      node match {
        case Relation(a: Aggregate) => {
          val r = a.mapCols { col =>
            max(l, min(u, col.expr)) AS col.alias }
          (r, ())
        }
        case _ => (node, ())
  } } }
```

CHORUS's rewriter API contains an embedded DSL for building SQL queries; the expression max(l, min(u, col.expr)) uses this embedded DSL to generate a new expression for the argument to the SUM aggregation. Running this rewriter on our simple query from earlier, with a lower bound of 0 and upper bound of 100, produces the following change:

```
SELECT SUM(trip_distance) FROM trips
                    ⇓
SELECT SUM(max(0, min(100, trip_distance))) AS sum
FROM trips
```

CHORUS's rewriting library is designed to address the challenges of rewriting tasks like the one above. Solving such tasks requires matching on generalized patterns in the query, and replacing sections of the query with new text. This process requires building an abstract syntax tree for the query, since searching through its text (e.g. with regular expressions) will not provide enough information about the query's structure to implement the correct semantics. Modern DBMSs use a variety of SQL dialects; many SQL queries in production are hundreds or thousands of lines long and use many of these special features, so analyzing these queries is difficult.

CHORUS provides a query parser based on Apache Calcite [1], and also allows extending the parser to support features of specialized dialects. We translate special features into an abstract syntax tree (AST) based on relational algebra for rewriting, and provide an API for pattern matching on pieces of the AST. The rewriteRecursive method, for example, makes it easy to search for a particular pattern in the query and replace it with something new. This method also allows the program to perform simultaneous analysis of the query, using the abstract interpretation framework described in the next section.

A second challenge of rewriting is producing *new* queries (or sections of queries). CHORUS provides an embedded domain-specific language (DSL) as a library of Scala operators for this purpose. SQL queries produced using our DSL have two major advantages over a simpler solution based on formatting strings: first, they are more readable, and can easily incorporate other AST nodes (as in min(u, col.expr) above), and second, the AST objects produced by our DSL operators can be easily translated *back* into different SQL dialects. For example, if the DBMS used for deployment provides the minimum function instead of min, CHORUS can translate the above query appropriately for this dialect.

4.2. Analysis

The next step is to analyze the rewritten query to determine its sensitivity. For this purpose, CHORUS provides a query analysis API which implements an abstract interpretation framework for SQL queries. Abstract interpretation is the execution of a program using *abstract values* instead of concrete (normal) values. Abstract values are members of an *abstract domain*, and each abstract value represents a set of possible concrete values. For example, the abstract domain {Even, Odd, Unknown} might correspond to the concrete domain of the integers, and the abstract value Even represents the set of even numbers. Abstract interpretation enables analysis of program properties: if a program outputs the value Even, then we know its output is always an even number. In CHORUS, we use abstract interpretation on SQL queries to determine properties like the sensitivity of the query independent of the underlying concrete data.

The sensitivity analysis for our rewritten query depends on three properties. First, the *stability* [55] of the underlying relation is 1. The stability of a relation measures how much the transformations used to create it from underlying tables change the number of rows before aggregation. A stability of 1 means that these transformations do not change the number of rows. Second, each value is clipped to lie between 0 and 100 before being aggregated. Third, the aggregation being performed is a SUM, whose global sensitivity is equal to $(u-l)\cdot s$ where u and l are the upper and lower clipping bounds, respectively, and s is the stability of the underlying relation.

We can use CHORUS to build an analysis which encodes these three properties by developing a pair of abstract domains that track properties of subexpressions of the query. The abstract values in these domains are placeholders for arbitrary relations, annotated with upper bounds on their stabilities or sensitivities. The *stability domain* tracks the stability (defined above) of relational expressions; the *sensitivity domain* tracks upper and lower bounds of values, and the global sensitivity (defined in Section 2) of values.

```
case class ColSens(
  sensitivity: Option[Double],
  upper: Double,
  lower: Double)

object SensDomain
  extends AbstractDomain[ColSens] {
  override val bottom: ColSens =
    ColSens(
      sensitivity = None,
      upper = PositiveInfinity,
      lower = NegativeInfinity) }
```

The second part is to implement the analysis itself, which encodes the rules for updating the domain. For stability, we will return a stability of 1 for relations resulting from a single table, and a stability of infinity otherwise. For sensitivity, we will follow the rules outlined above: the sensitivity and upper and lower bounds of a column reference start out infinite; upper and lower bounds are introduced using the max and min functions, respectively; and sensitivity is bounded using the COUNT and SUM aggregation functions. The definition of the analysis appears in Figure 2.

The abstract interpretation framework in CHORUS addresses a key challenge of deploying differentially private mechanisms for realistic SQL queries: SQL queries can be extremely complicated, and building sound analyses of these queries (e.g. for determining sensitivity) is correspondingly difficult.

The CHORUS analysis library allows a small amount of code to define sound analyses which support these complicated queries. The analysis we have defined here can be used even on queries with features we have not discussed, including subqueries, queries with WHERE clauses, and so on. This ability is based on the same parser and relational abstract syntax tree support described in the last section, extended with a dataflow analysis.

CHORUS implements a dataflow library for relational ASTs based on the classic monotone framework [45]. In this framework, the programmer defines an abstract domain as a lattice, provides a *join* operator for computing the least upper bound of two points in the lattice, and writes *transfer functions* to specify the effect of each

```
class SensAnalysis extends ColAnalysis(SensDomain) {
  override def transferAggregate(
      node: Aggregate,
      state: ColSens) = {
    node.aggFunction match {
      case Some(func) =>
        val newSensitivity = func match {
          case COUNT => state.nodeFact.stability
          case SUM   =>
            (state.nodeFact.upper    state.nodeFact.lower)
            * state.nodeFact.stability
          case _     => PositiveInfinity
        }

        colState.copy(
          sensitivity = Some(newSensitivity),
          upper = PositiveInfinity,
          lower = NegativeInfinity)
    } }

  override def transferExpression(
      node: RexNode,
      state: ColSens) = {
    val bot = state.copy(
      sensitivity = None,
      upper = PositiveInfinity,
      lower = PositiveInfinity)

    node match {
      case c: RexCall =>
        c.getOperator.getKind match {
          case SqlKind.Max =>
            state.copy(upper=c.getOperands[0])
          case SqlKind.Min =>
            state.copy(lower=c.getOperands[0])
          case _ => bot
        }
      case _ => bot
    } } }
```

Figure 2: Static Analysis for Determining Sensitivity of SUM and COUNT Queries, with Clipping

operator in the programming language on its inputs. In CHORUS, these are specified as regular Scala functions. The framework uses these to generate an abstract interpreter [63] that checks properties of programs by computing conservative approximations of the program's output.

This abstract interpretation framework greatly simplifies the task of analyzing a query to determine its properties. Above, we have defined a domain which tracks *two* properties of the query: (1) bounded ranges for values in sub-parts of the query, introduced by clipping, and (2) sensitivity of the query, based on aggregation functions and the bounded ranges determined in part (1). Implemented directly on abstract syntax trees, such an analysis would comprise hundreds of lines of complicated code; CHORUS's analysis library is designed to provide re-usable components to make these tasks simple.

4.3. Post-Processing

Finally, we use the rewriter and analyzer we have defined to produce a new SQL query with bounded sensitivity, run the query using the DBMS, and add Laplace noise to the results.

```
def laplaceMechClip(query: String,
    epsilon: Double): List[Row] = {
  val rewritten = rewriteClip(0, 100)(query)
  val sens = SensAnalysis().analyze(rewritten)

  val r = DB.execute(rewritten)
  r + Utils.Laplace(sens / epsilon)
}
```

Here, the call to DB.execute actually runs the query on the DBMS used in deployment. CHORUS works with any DBMS with a JDBC interface, and the configuration information for the DBMS is specified in CHORUS's configuration file.

This mechanism is ready for deployment using CHORUS, which enables it to run alongside any standard SQL DBMS and scale to massive databases. Since the code we have defined here is part of a *static* analysis of just the query, the scalability of our mechanism depends only on the ability of the cooperating DBMS to execute the rewritten query (which has only minor changes).

Our mechanism definitions are simply Scala functions, and can be exposed to analysts in a number of different ways depending on the deployment scenario (more in Section 6). The CHORUS post-processing library provides a number of useful utilities, including the Laplace mechanism (used above), the Gaussian mechanism, the Exponential mechanism, and various forms of clipping.

This simple example illustrates how CHORUS enables a *rewrite-analyze-postprocess* pattern to achieve its design goals of enabling mechanisms which change how the query executes while integrating easily with an existing DBMS. We will see this pattern repeatedly in the more complicated mechanisms we will develop in Section 5.

4.4. Budgeting & Mechanism Selection

The CHORUS API provides two interfaces for privacy budget accounting: PrivacyCost, to represent privacy costs, and PrivacyAccountant, to track the total cost of composing many mechanisms.

The PrivacyCost interface requires the programmer to define the + method in accordance with the sequential composition property of the corresponding privacy definition. The following two classes define privacy costs for pure ϵ-differential privacy and Rényi differential privacy [58] (we have also defined privacy cost for (ϵ, δ)-differential privacy and Zero-concentrated differential privacy [22]):

```
case class EpsilonDPCost(epsilon: Double)
    extends PrivacyCost {
  def +(other: PrivacyCost) = other match {
    case EpsilonDPCost(otherEpsilon) =>
      EpsilonDPCost(epsilon + otherEpsilon) }}

case class RenyiDPCost(alpha: Int, epsilon: Double)
    extends PrivacyCost {
  def +(other: PrivacyCost) = other match {
    case RenyiDPCost(otherAlpha, otherEpsilon) =>
      RenyiDPCost(math.max(alpha, otherAlpha),
                  epsilon + otherEpsilon) }}
```

The PrivacyAccountant class enables different approaches to computing the total budget used over many queries. The base class tracks the privacy costs of individual mechanisms, and the programmer defines a getTotalCost method to compose these costs. For example, the following two classes define accountants for advanced composition of pure ϵ-differentially private mechanisms and for Rényi differential privacy:

```
class AdvancedCompositionAccountant(delta: Double)
    extends PrivacyAccountant {
  def getTotalCost() = {
    val epsilons = costs.map(_.epsilon)
    val totalEpsilon = 2*(epsilons.max)*
      math.sqrt(2*(epsilons.length)*math.log(1/delta))
    EpsilonDeltaDPCost(totalEpsilon, delta) }}

class RenyiCompositionAccountant
    extends PrivacyAccountant {
  def getTotalCost() =
    costs.fold(RenyiDPCost(0, 0))(_ + _) }
```

Finally, the ChorusMechanism abstract class defines a standard interface for mechanisms, and integrates them with a chosen privacy accountant. Each mechanism class defines a run method that returns a differentially private result and a PrivacyCost object. To run a mechanism with accountant a, the programmer calls execute(a), which invokes run, adds the mechanism's cost to the accountant, and returns the result. For example, we can package our Laplace mechanism into a CHORUS mechanism:

```
class laplaceMechanism(epsilon: Double, l: Double,
  u: Double, root: Relation)
    extends ChorusMechanism[List[DB.Row]] {
  def run() = (laplaceMechClip(epsilon, l, u, root),
               EpsilonDPCost(epsilon)) }
```

These interfaces provide a flexible way to build systems that leverage existing mechanisms, permit building new mechanisms from old ones, and even allow implementing automatic mechanism selection (via "mechanisms" that use query properties to select from a list of mechanisms to run). For example, for our prototype deployment, we implemented a CHORUS mechanism that runs one of three individual mechanisms using a simple rule-based approach.

5. Mechanism Development with CHORUS

This section demonstrates the use of CHORUS to implement a number of different mechanisms, from simple ones based on the Laplace mechanism to more advanced algorithms like Sample and Aggregate and MWEM.

5.1. Average Queries

Average queries are typically answered with differential privacy by transforming the query into two separate differentially private queries—a SUM and a COUNT—and performing the division as post-processing.

We can implement this approach in CHORUS by reusing the building blocks we have already built. First, we will develop two rewriters: one that turns AVG into SUM, and another that turns AVG into COUNT. We will rewrite the input query twice, once with each rewriter, and then run both rewritten queries using the mechanism we developed in Section 4. The full implementation appears in Figure 3.

This modular approach illustrates a key benefit of CHORUS's design: the ability to re-use existing mechanisms to implement new ones. We will see this pattern used again in later mechanisms to reduce complexity.

5.2. Report Noisy Max

The *report noisy max* mechanism [32] takes a list of queries as input, adds independently drawn noisy to each one, and returns the *index* of the maximum noisy result. The key advantage of report noisy max is that it consumes privacy budget proportional to *one query*, regardless of the length of the list of queries specified by the analyst.

This mechanism can be implemented in CHORUS by extending the ideas in the laplaceMechClip mechanism to a list of queries, and returning only the index of the maximum value in the resulting list:

```
def avgMech(query: Relation, epsilon: Double) = {
  def replaceAvg(q: Relation, aggFn: Aggregate) = {
    Chorus.recursiveRewrite(q) { (node: Relation) =>
      node match {
        case Relation(a: SqlAvgAggFunction) => {
          val r = a.mapCols { col =>
            aggFn(col) AS col.alias }
          (r, ())
        }
        case _ => (node, ())
  } } }

  val sumQuery = replaceAvg(query, Sum)
  val countQuery = replaceAvg(query, Count)

  val r1 = laplaceMechClip(sumQuery, epsilon / 2)
  val r2 = laplaceMechClip(countQuery, epsilon / 2)
  r1 / r2 }
```

Figure 3: AVG Mechanism in CHORUS

```
def reportNoisyMax(queries: List[Relation],
  epsilon: Double): Int = {
    val results = queries.map(
      laplaceMechClip(_, epsilon, 0, 1))
    Utils.argmax(results) }
```

Our implementation of the report noisy max mechanism demonstrates two important principles. First is the use of a workload of queries: since CHORUS mechanisms are implemented as regular Scala functions, they can accept queries in any format, including a workload of SQL queries. Second is the splitting of mechanism logic between Scala and SQL: query results can be post-processed with Scala code in arbitrary ways (here, we use Scala to find the maximum workload result).

5.3. Exponential Mechanism

The *exponential mechanism* [54] is the generalized version of report noisy max: it selects an element of a set R which approximately maximizes the value of a *scoring function*. The scoring function $q : D^n \to R \to \mathbb{R}$ assigns numeric scores to each element of R based on the private database (in report noisy max, the scoring function simply returns the value of the query). For a scoring function with sensitivity $\Delta = GS(q)$ and a database X, the exponential mechanism outputs $r \sim R$ with probability proportional to:

$$\Pr[r] \sim \exp\left(\frac{\epsilon q(X, r)}{2\Delta}\right)$$

In report noisy max, the scoring function was fixed, so it was easy to assume something about its sensitivity and apply it directly to each query in the workload. The generalized exponential mechanism makes this process more difficult: since the analyst specifies the scoring function, we cannot assume anything about its sensitivity.

The solution is to ask the analyst to provide the scoring function *as a query*, so that we can analyze its sensitivity directly. This query should produce a relation mapping elements in R to their scores. For example, the following query implements a scoring function which will allow selecting the day of the week with approximately the largest number of trips:

```
SELECT day, COUNT(*) as score
FROM trips
GROUP BY day
```

We can use the same basic analysis as before to show that this query has a sensitivity of 1, and then perform the selection step of the exponential mechanism in a post-processing step.

```
def exponentialMech(scoring: Relation,
                    epsilon: Double) = {
  val s = SensAnalysis.analyze(scoring)
  val scores = DB.execute(scoring)
  val totalScore = scores.map(_._2).sum()
  val probabilities = scores.map { r =>
    (r(0),
     (epsilon * (r(1) / totalScore)) / (2*s) ) }

  Utils.chooseWithProbability(probabilities)._1
}
```

Our implementation here uses `Utils.chooseWithProbability` to select from the elements of R with the appropriate probabilities (this function expects a list of tuples mapping domain elements to their probabilities, and implements weighted random selection). We return just the identity of the element which was selected, similar to the report noisy max mechanism.

```
def sparseVectorMech(queries: List[Relation],
    threshold: Double, epsilon: Double) = {
  val sens = rewritten.map(SensAnalysis().analyze(_))

  // require sensitivity 1 queries
  if (sens.exists(_ > 1))
    return None

  // generate noisy threshold
  val T = threshold + Utils.laplaceSample(2/epsilon)

  for (i < 0 to queries.length()) {
    val r = DB.execute(queries(i))
    if (r + Utils.Laplace(4/epsilon) >= T)
      return Some(i)  }

  return None }
```

Figure 4: Sparse Vector Technique in CHORUS

5.4. Sparse Vector Technique

The *sparse vector technique* (SVT) [32] releases the index (but not the result) of the first query in a sequence of queries whose result exceeds a threshold set by the analyst. Like report noisy max, SVT consumes privacy budget proportional to just one query. In situations where only a small number of queries are likely to have large enough results to be useful to the analyst (but the analyst does not know which ones), SVT can be applied repeatedly to answer the useful queries while minimizing privacy budget consumption.

We can define a CHORUS mechanism to implement SVT in a similar way to the report noisy max mechanism. In contrast to the report noisy max mechanism, however, SVT is *iterative*—it runs the queries in the workload in sequence, and may halt before running all of them. SVT first adds Laplace noise to the threshold, then compares the noisy result of each query to the noisy threshold. SVT releases the index of the first query whose noisy value exceeds the noisy threshold.

Our implementation appears in Figure 4. First, it analyzes each query to ensure that the query's sensitivity does not exceed 1, and returns `None` if not. Then, it uses `Utils.Laplace` to add noise to both the threshold and each query result, and returns `Some(i)` for the first index `i` whose noisy query result exceeds the noisy threshold. If no query exceeds the threshold, we return `None`.

This implementation corresponds to the AboveThreshold algorithm described by Dwork and Roth [32]. This algorithm can be combined with our earlier mechanisms to release the *value* of the first query above the threshold:

```
def sparseVectorMechValue(queries: List[Relation],
    thresh: Double, epsilon: Double) = {
  val i = sparseVectorMech(queries, thresh, epsilon/2)
  laplaceMechClip(queries(i), epsilon/2) }
```

This mechanism combines SVT with the Laplace mechanism to find the value of the first query above the threshold (not just its index), splitting the privacy budget between these two tasks. This mechanism illustrates the ease of combining mechanisms in CHORUS to build complex functionality.

5.5. Sample & Aggregate

The Sample & Aggregate [64], [75] mechanism works for all statistical estimators, but does not support joins. Sample & Aggregate has been implemented in GUPT [59], a standalone data processing engine that operates on Python programs, but has never been integrated with a high-performance DBMS. As defined by Smith [75], the mechanism has three steps:

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

We can use the DBMS to accomplish tasks 1 and 2 by modifying the analyst's original query. We add a GROUP BY clause to the original query which groups rows according to their row number. Sample and aggregate does not require the subsamples to be randomized, so basing the selection of the subsamples on row number satisfies its requirements. For example, we can transform a simple AVG query as follows:

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

This transformation generates n subsamples and runs the original query on each one. Once we have obtained the answers to query on the subsamples, we can perform differentially private aggregation as a post-processing step—with clipping followed by a noisy average. The complete implementation appears in Figure 5.

5.6. Additional Mechanisms

We present CHORUS implementations of three additional mechanisms in in Appendix A: Weighted PINQ, the Matrix Mechanism, and MWEM.

6. Implementation & Deployment

This section describes the deployment of CHORUS to protect sensitive data and provide a secure, differentially private interface for analysts to query that data. We also describe our experience deploying CHORUS to enforce differential privacy at Uber.

A typical deployment of a CHORUS mechanism includes a centralized query interface which allows the

```
def rewriteSAA(n: Int, root: Relation) = {
  root.rewriteRecursive(UnitDomain) {
    (node, orig, _) =>
      node match {
        // Add new subsample number column
        case Relation(t: TableScan) =>
          (node.project(*, RowNumMod AS "_grp"), ())

        // Group by the subsample number
        case Relation(a: Aggregate) =>
          (a addGroupedColumn col("_grp"), ())

        case _ => (node, ())
  } } }

def saaMech(query: Relation, l: Double, u: Double,
    numSubsamples: Int, epsilon: Double) = {
  // rewrite the query to perform subsampling
  val rewritten = rewriteSAA(numSubsamples, query)
  // execute rewritten query and get subsample results
  val r = db.execute(rewritten).map(_._2)

  // calculate sensitivity
  val sens = (u   l) / numSubsamples

  // calculate noisy average via clipping
  val mean = Utils.mean(Utils.clip(r, l, u))
  r + Utils.Laplace(sens / epsilon) }
```

Figure 5: Sample & Aggregate in CHORUS

DBMSs containing sensitive data to be queried *only* via the mechanism. In such a deployment, untrusted analysts submit queries to the centralized query interface, which runs the mechanism and updates the privacy budget. The interface may enable auditing of the budget by a trusted curator of the system, and rejection of queries after the budget has been exhausted.

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

The approach could also be implemented with other query optimization frameworks or rule-based query rewriters such as Starburst [66], ORCA [5], and Cascades [36].

Real-world Deployment. CHORUS has been tested in a deployment to enforce differential privacy for queries over customer data at Uber. The primary goals of this deployment are to protect the privacy of customers from insider attacks, and to ensure compliance with the requirements of Europe's General Data Protection Regulation (GDPR) [4]. The mode of deployment calls for CHORUS to process more than 10,000 queries per day.

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

Deployment involved building a minimal wrapper around the CHORUS library to expose its rewriting functionality as a microservice. The only required change to the data environment was a single modification to

the query interface, to submit queries to the CHORUS microservice for rewriting before execution. The wrapper around CHORUS also queries a policy microservice to determine the security and privacy policy for the user submitting the query. This policy informs which mechanism is used—by default, differential privacy is required, but for some privileged users performing specific business tasks, differential privacy is only used for older data.

A major challenge of this deployment has been supporting the variety of SQL dialects used by the various DBMSs. This challenge motivated the built-in support for different dialects in the CHORUS framework.

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

7. Evaluation

In this section, we evaluate the performance overhead of enforcing differential privacy using CHORUS in the context of real-world queries on a large production dataset. Our results demonstrate that the mechanism implementations we have developed using CHORUS scale effectively to realistic datasets using commodity DBMSs.

Our evaluation is intended to demonstrate the scalability of the CHORUS approach, rather than the ability of the mechanisms we implemented to produce accurate results. The accuracy we obtain in each of our experiments is a direct result of the underlying mechanism used in the experiment, and the same accuracy would be obtained using an alternative implementation of the same mechanism.

Corpus. We use a corpus of 18,774 real-world queries containing all statistical queries executed by data analysts at Uber during a single month. The corpus includes queries written for several use cases including fraud detection, marketing, business intelligence and general data exploration.

Dataset. We used a database of data sampled from the production database in our evaluation. This database contained 300 million records representing trip data similar in nature to the New York City Taxi Trip Data [6]—including information about trips, riders, and drivers.

Mechanisms. In our evaluation, we considered two more complicated variants of the sensitivity-based mechanism we developed in Section 4: elastic sensitivity [44] and restricted sensitivity [19]. These mechanisms can answer existing counting and summation queries written by analysts unfamiliar with differential privacy, like those in our corpus. We also considered the performance of Weighted PINQ [67], since it performs more serious modifications to the analyst's query.

7.1. Performance Overhead

We conduct a performance evaluation demonstrating the performance overhead of several mechanisms implemented with CHORUS.

Experiment Setup. We used a single HP Vertica 7.2.3 [7] node containing 300 million records including trips, rider and driver information and other associated data stored across 8 tables. We submitted the queries locally and ran queries sequentially to avoid any effects from network latency and concurrent workloads.

To establish a baseline we ran each original query 10 times and recorded the average after dropping the lowest and highest times to control for outliers. Then, we ran each CHORUS mechanism 10 times and recorded the average execution time, again dropping the fastest and slowest times. We calculate the overhead for each query by comparing the average runtime of the original query and the CHORUS mechanism.

Results. Figure 6 shows the distribution of overhead as a function of original query execution time. This distribution shows that the percentage overhead is highest when the original query was very fast (less than 100ms). This is because even a small incremental performance cost is fractionally larger for these queries.

WPINQ significantly alters the way the query executes (see Section 5) and these changes increase query execution time. In particular, the query transformation adds a new join to the query each time weights are rescaled (i.e. one new join for each join in the original query), and these new joins result in the additional overhead. Figure 6 shows that, in both cases, the performance impact is amortized over higher query execution times, resulting in a lower relative overhead for more expensive queries.

7.2. Utility

We also measured the ability of some of the mechanisms we have implemented with CHORUS to produce accurate differentially private results for queries in our corpus, primarily as a study of whether or not differential privacy is a good fit for these queries. These mechanisms were previously proposed, and their accuracy properties were previously known. Our experimental results confirm existing knowledge using real-world queries and data.

For this experiment, we used the elastic sensitivity and restricted sensitivity mechanisms described earlier. Evaluating the utility of more complicated mechanisms would require re-formulating the queries in our corpus with the help of the original analyst. All of these mechanisms have been previously evaluated on synthetic query workloads, and their ability to improve utility is well-understood; as we gain experience with practical deployments, analysts will begin to adopt these mechanisms and re-formulate their queries.

Experiment Setup. We use the same setup described in the previous section. For each query, we set the privacy budget $\epsilon = 0.1$ for all mechanisms. For Elastic Sensitivity, we set $\delta = \frac{1}{n^2}$ (where n is the database size).

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

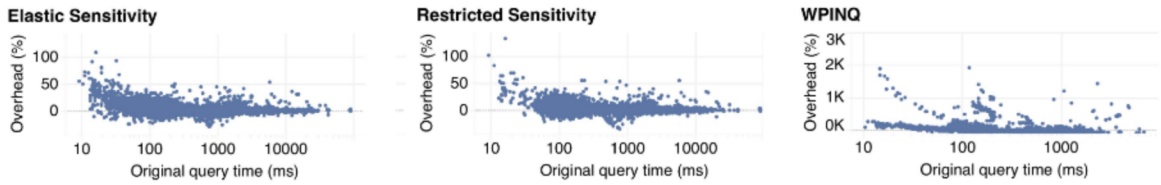

Figure 6: Performance overhead of differential privacy mechanisms by execution time of original query.

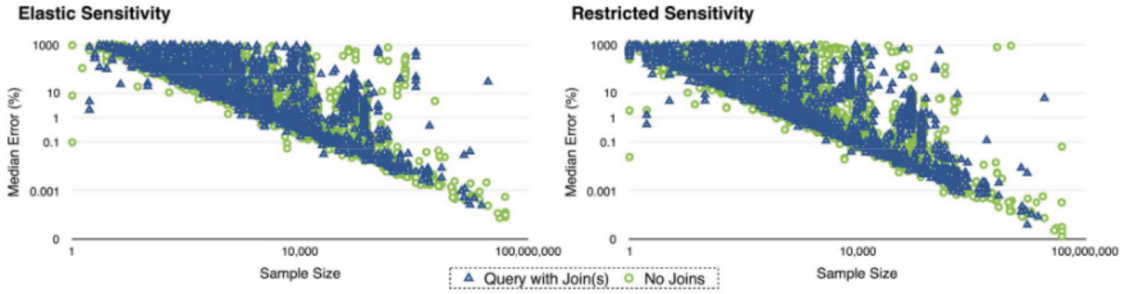

Figure 7: Utility of elastic sensitivity and restricted sensitivity, by presence of joins.

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

Results. Figure 7 shows the results of this experiment. Both mechanisms exhibit the expected inverse relationship between sample size and error; moreover, this trend is apparent for queries with and without joins.

Importantly, a very large portion of the queries with large sample sizes have very small relative error—the majority of queries with a sample size of over 10,000, for example, have less than 1% error. This suggests that for at least a subset of the queries in this production corpus, differential privacy can be enforced without significant harm to accuracy. Furthermore, more advanced mechanisms like the ones described in Section 5 could provide significant improvements for the accuracy of these queries.

A significant number of queries with small sample size also result in large relative error. It is likely that many of these queries are *intended* to violate privacy—perhaps they examine the data of an individual or small set of individuals directly—and so differential privacy is probably not appropriate for these queries.

7.3. Discussion and Key Takeaways

Strengths & weaknesses of differential privacy. The mechanisms we studied generally worked best for statistical queries over large populations. None of the mechanisms was able to provide accurate results (e.g. within 1% error) for a significant number of queries over populations smaller than 1,000. These results confirm the existing wisdom that differential privacy is ill-suited for queries with small sample sizes. For large populations (e.g. more than 10,000), accurate differentially private results for real-world queries appears to be an achievable goal. A large set of such queries exists in our corpus. These results suggest that differential privacy can provide both strong privacy guarantees and accurate query responses for a large portion of the queries written by analysts in practice.

Our results also agree with the prior knowledge that queries with joins make ensuring differential privacy more challenging. For both mechanisms we used in our evaluation, the proportion of queries with less than 1% error was much smaller for queries with joins than for queries without joins. Differential privacy for joins is an active area of research [44], [49], and we hope that future mechanisms can be implemented in CHORUS to provide more accurate answers for these queries.

Mechanism performance. Our performance evaluation demonstrates the scalability of mechanisms implemented with CHORUS—the vast majority of the queries executed with the elastic sensitivity and restricted sensitivity mechanisms resulted in less than 50% overhead, and the mean overhead for both was below 25%. However, the results also highlight the variability in computation costs of differential privacy mechanisms—WPINQ's added joins resulted in high performance overhead for some queries.

8. Related Work

Differential Privacy. Differential privacy was originally proposed by Dwork [28]–[30]. The reference by Dwork and Roth [32] provides an overview of the field.

Much recent work has focused on task-specific mechanisms for graph analysis [24], [42], [46], [47], [72], range

queries [8], [25], [41], [50]–[53], [68], [81], [82], [85], and analysis of data streams [31], [74]. As described in Section 7.3, such mechanisms are complementary to our approach, and could be implemented on top of CHORUS to provide more efficient use of the privacy budget.

Differential Privacy Systems. As differential privacy is more widely adopted, scalable implementations of differential privacy mechanisms have received more attention. Wilson et al. recently developed an open source, highly performant C++ library [80] which provides basic differential privacy mechanisms as an extension to PostgreSQL. However, their approach does not support other DBMSs. It also does not provide a framework for development of additional mechanisms; more complex mechanisms, like MWEM, would need to be added directly to the C++ library as new primitives.

The problem of answering SQL queries has also received considerable attention in recent years. The FLEX [44] system answers counting queries with differential privacy, and since it implements the post-processing architecture, it is DBMS-independent. However, FLEX cannot implement more complex mechanisms like MWEM. PrivateSQL [49] answers SQL queries with differential privacy by generating differentially private *synopses* of views, then using the synopses to answer queries. This approach is potentially scalable, but limits the set of implementable mechanisms.

A number of other systems for enforcing differential privacy have been developed. PINQ [55] supports a LINQ-based query language, and implements the Laplace mechanism with a measure of global sensitivity. Weighted PINQ [67] extends PINQ to weighted datasets, and implements a specialized mechanism for that setting.

Airavat [71] enforces differential privacy for MapReduce programs using the Laplace mechanism. Fuzz [35], [40] enforces differential privacy for functional programs, using the Laplace mechanism in an approach similar to PINQ. DJoin [60] enforces differential privacy for queries over distributed datasets. Due to the additional restrictions associated with this setting, DJoin requires the use of special cryptographic functions during query execution so is incompatible with existing databases. GUPT [59] implements the Sample & Aggregate framework for Python programs. None of these systems offers integration with high-performance DBMSs.

Security & Privacy via Query Rewriting. Automated query transformations have been used in previous work to implement access control. Stonebreaker and Wong [76] presented the first approach. Barker and Rosenthal [11] extended the approach to role-based access control by first constructing a view that encodes the access control policy, then rewriting input queries to add `WHERE` clauses that query the view. Byun and Li [23] use a similar approach to enforce purpose-based access control: purposes are attached to data in the database, then queries are modified to enforce purpose restrictions drawn from a policy.

Since then, a number of rewriting-based approaches have been proposed for enforcing access control. Agrawal et al. [9] use query rewriting to enforce row-level privacy policies focused on access control. Bender et al. [17], [18] combine query rewriting with specially-designed views to enforce privacy policies organized in "disclosure lattices." Wang et al. [78] propose fine-grained cell-level access control policies, with an enforcement mechanism based on query rewriting. Rizvi et al. [70] propose an access control mechanism based on "authorization views" which define policies, and use query rewriting to re-phrase queries in terms of these views. Oracle's Virtual Private Database [20] enforces fine-grained access control policies using query rewriting. All of these are primarily focused on access control policies.

Mehta et al. present Qapla [56], a system which uses query rewriting to enforce policies written in SQL. Notably, Qapla includes *aggregation policies*, which go beyond traditional access control policies to allow the release of aggregate statistics while protecting the underlying rows, but Qapla does not support differential privacy or other formal notions of privacy.

Guarnieri et al. [38], [39] explore the challenges of enforcing access control policies in the context of a complicated and expressive query language like SQL, and highlight the need for provable guarantees about the enforcement mechanism. Zhang and Mendelzon [86] study one of these challanges—the problem of "query containment"—in the context of proving correctness for query rewriting enforcement mechanisms. These results reinforce the value of additional work in the future to verify the correctness of our rewriting algorithms.

9. Conclusion

This paper presents CHORUS, a framework which enables a novel cooperative architecture for enforcing differential privacy. CHORUS works closely with a high-performance DBMS to scale differential privacy mechanisms to real-world deployments. CHORUS combines the strengths of integrated implementations (whose scalability does not match high-performance industrial DBMSs) and post-processing based implementations (which scale up, but are incompatible with many modern differential privacy mechanisms). We have described how CHORUS can be deployed to provide differential privacy, and released it as open source [3].

Acknowledgments

The authors would like to thank Om Thakkar and the anonymous reviewers for their helpful comments, and Uber's Privacy Engineering team for collaboration on this project. This work was supported by the Center for Long-Term Cybersecurity, and DARPA & SPAWAR under contract N66001-15-C-4066. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

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

[2] 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. Accessed: 2016-08-14.

[3] Chorus Source Code. https://github.com/uvm-plaid/chorus.

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

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

[6] TLC Trip Record Data. http://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2020-03-21.

[7] Vertica Advanced Analytics. http://www.vertica.com. Accessed: 2020-01-12.

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

[9] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1013–1022. IEEE, 2005.

[10] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. *PACMPL*, 2(POPL):58:1–58:30, 2018.

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

[12] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Strub. Proving differential privacy in hoare logic. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 411–424, July 2014.

[13] Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. Relational ⋆⋆\star-liftings for differential privacy. *Logical Methods in Computer Science*, 15(4), 2019.

[14] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, page 749–758, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 97–110, New York, NY, USA, 2012. Association for Computing Machinery.

[16] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3), November 2013.

[17] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1411–1422, 2014.

[18] Gabriel M Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 869–880, 2013.

[19] Jeremiah Blocki, Avrim Blum, Anupam Datta, and Or 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.

[20] Kristy Browder and Mary Ann Davidson. The virtual private database in oracle9ir2. *Oracle Technical White Paper, Oracle Corporation*, 500:280, 2002.

[21] Mark Bun, Cynthia Dwork, Guy N Rothblum, and Thomas Steinke. Composable and versatile privacy via truncated cdp. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 74–86. ACM, 2018.

[22] Mark Bun and Thomas Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*, pages 635–658. Springer, 2016.

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

[24] Shixi Chen and Shuigeng 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.

[25] Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, Entong Shen, and Ting Yu. Differentially private spatial decompositions. In *Data engineering (ICDE), 2012 IEEE 28th international conference on*, pages 20–31. IEEE, 2012.

[26] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. Probabilistic relational reasoning via metrics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–19. IEEE, 2019.

[27] Yves-Alexandre de Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3, 2013.

[28] Cynthia Dwork. Differential privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2006.

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

[30] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[31] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N Rothblum. Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 715–724. ACM, 2010.

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

[33] Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. Boosting and differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 51–60. IEEE, 2010.

[34] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra 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.

[35] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN Notices*, volume 48, pages 357–370. ACM, 2013.

[36] Goetz Graefe. The cascades framework for query optimization.

[37] Paolo Guagliardo and Leonid Libkin. A formal semantics of sql queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39, 2017.

[38] Marco Guarnieri and David Basin. Optimal security-aware query processing. *Proceedings of the VLDB Endowment*, 7(12):1307–1318, 2014.

[39] Marco Guarnieri, Srdjan Marinovic, and David Basin. Strong and provably secure database access control. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 163–178. IEEE, 2016.

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

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

[42] Michael Hay, Chao Li, Gerome Miklau, and David Jensen. Accurate estimation of the degree distribution of private networks. In *Data Mining, 2009. ICDM 09. Ninth IEEE International Conference on*, pages 169–178. IEEE, 2009.

[43] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, and Dan Zhang. Principled evaluation of differentially private algorithms using dpbench. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 139–154. ACM, 2016.

[44] Noah Johnson, Joseph P Near, and Dawn Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

[45] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. 7(3):305–317, January 1977. Data Flow Analysis.

[46] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. *Proceedings of the VLDB Endowment*, 4(11):1146–1157, 2011.

[47] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Analyzing graphs with node differential privacy. In *Theory of Cryptography*, pages 457–476. Springer, 2013.

[48] Ios Kotsogiannis, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. Pythia: Data dependent differentially private algorithm selection. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1323–1337. ACM, 2017.

[49] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. Privatesql: a differentially private sql query engine. *Proceedings of the VLDB Endowment*, 12(11):1371–1384, 2019.

[50] Chao Li, Michael Hay, Gerome Miklau, and Yue Wang. A data- and workload-aware algorithm for range queries under differential privacy. *Proceedings of the VLDB Endowment*, 7(5):341–352, 2014.

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

[52] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. The matrix mechanism: optimizing linear counting queries under differential privacy. *The VLDB Journal*, 24(6):757–781, 2015.

[53] Ryan McKenna, Gerome Miklau, Michael Hay, and Ashwin Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *Proceedings of the VLDB Endowment*, 11(10):1206–1219, 2018.

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

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

[56] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1463–1479, 2017.

[57] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012.

[58] Ilya Mironov. Rényi differential privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 263–275. IEEE, 2017.

[59] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David 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.

[60] Arjun Narayan and Andreas 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.

[61] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the Netflix prize dataset. *CoRR*, abs/cs/0610105, 2006.

[62] Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[63] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[64] Kobbi Nissim, Sofya Raskhodnikova, and Adam 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.

[65] Vijay Pandurangan. On taxis and rainbows: Lessons from NYC's improperly anonymized taxi logs. https://medium.com/@vijayp/of-taxis-and-rainbows-f6bc289679a1. Accessed: 2015-11-09.

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

[67] Davide Proserpio, Sharon Goldberg, and Frank 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.

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

[69] Jason Reed and Benjamin C Pierce. Distance makes the types grow stronger: a calculus for differential privacy. *ACM Sigplan Notices*, 45(9):157–168, 2010.

[70] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562, 2004.

[71] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, volume 10, pages 297–312, 2010.

[72] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben 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.

[73] T. Sato, G. Barthe, M. Gaboardi, J. Hsu, and S. Katsumata. Approximate span liftings: Compositional semantics for relaxations of differential privacy. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14, June 2019.

[74] Elaine Shi, HTH Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium (NDSS)*. Internet Society., 2011.

[75] Adam 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.

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

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

[78] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd international conference on Very large data bases*, pages 555–566. VLDB Endowment, 2007.

[79] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–669, 2019.

[80] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. Differentially private sql with bounded user contribution. *arXiv preprint arXiv:1909.01917*, 2019.

[81] Yonghui Xiao, Li Xiong, Liyue Fan, and Slawomir Goryczka. Dpcube: differentially private histogram release through multidimensional partitioning. *arXiv preprint arXiv:1202.5358*, 2012.

[82] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, Ge Yu, and Marianne Winslett. Differentially private histogram publication. *The VLDB Journal*, 22(6):797–822, 2013.

[83] Danfeng Zhang and Daniel Kifer. Lightdp: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 888–901, 2017.

[84] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C Pierce, and Aaron Roth. Fuzzi: A three-level logic for differential privacy. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–28, 2019.

[85] Xiaojian Zhang, Rui Chen, Jianliang Xu, Xiaofeng Meng, and Yingtao Xie. Towards accurate histogram publication under differential privacy. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 587–595. SIAM, 2014.

[86] Zheng Zhang and Alberto O Mendelzon. Authorization views and conditional query containment. In *International Conference on Database Theory*, pages 259–273. Springer, 2005.

Appendix A. Additional Mechanisms

This appendix presents CHORUS implementations of three additional mechanisms: Weighted PINQ (Appendix A.1), the Matrix Mechanism (Appendix A.2), and MWEM (Appendix A.3).

A.1. Weighted PINQ

Weighted PINQ (WPINQ) enforces differential privacy for counting queries with equijoins. A key distinction of this mechanism is that it produces a differentially private *metric* (called a *weight*), rather than a count. These weights are suitable for use in a workflow that generates differentially private synthetic data, from which counts are easily derived. The workflow described in [67] uses weights as input to a Markov chain Monte Carlo (MCMC) process.

Our CHORUS implementation of WPINQ computes noisy weights for a given counting query according to the mechanism's definition [67]. Since the weights are differentially private, they can be released to the analyst for use with any desired workflow.

The WPINQ mechanism adds a weight to each row of the database, updates the weights as the query executes to ensure that the query has a sensitivity of 1, and uses the Laplace mechanism to add noise to the weighted query result. WPINQ has been implemented as a standalone data processing engine with a specialized query language, but since the mechanism cannot be implemented via post-processing alone, it has not been integrated into any SQL DBMS.

Where a standard database is a collection of tuples in D^n, a weighted database (as defined in Proserpio et al. [67]) is a function from a tuple to its weight ($D \to \mathbb{R}$). In this setting, counting the number of tuples with a particular property is analogous to summing the weights of all such tuples. Counting queries can therefore be performed using summations.

In fact, summing weights in a weighted dataset produces exactly the same result as the corresponding counting query on the original dataset, when the query does not contain joins. When the query does contain joins, WPINQ scales the weight of each row of the join's output to maintain a sensitivity of 1. Proserpio et al. [67] define the weight of each row in a join as follows, where A_k is the weights of rows of relation A with join key k:

$$A \bowtie B = \sum_k \frac{A_k \times B'_k}{A_k + B_k} \tag{1}$$

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

In our CHORUS implementation of WPINQ, we use the DBMS to track the weights associated with each column in computed relations. We can accomplish this by modifying the analyst's query to add weight a column to each relation. Consider the transformation for a simple counting query, in which we initialize each weight to 1:

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

This transformation adds a weight of 1 to each row in the table, and changes the COUNT aggregation function into a SUM of the rows' weights. The correctness of this transformation is easy to see: as required by WPINQ [67], the transformed query adds a weight to each row, and uses SUM in place of COUNT.

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

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

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

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

The correctness of this transformation follows from equation (1). The relation tnorms corresponds to A_k, and dnorms to B_k. For each key, t.weight corresponds to A_k, and d.weight to B_k.

Finally, we can accomplish the third task (adding Laplace noise scaled to $1/\epsilon$) as a post-processing task. Our complete CHORUS implementation defines a recursive rewriter that replaces table references with subqueries that

```
def matrixMech(workload: Matrix[Int],
               strategyMat: Matrix[Int],
               strategyQs: List[String],
               epsilon: Double) = {
  // answer the queries in the cell list
  val answers = strategyQs.map { q =>
    laplaceMechClip(q, epsilon / strategyQs.length()) }

  workload times (strategy.mpInverse()
                  times colMatrix(answers))
}
```

Figure 8: The Matrix Mechanism in CHORUS

initialize weights to 1, and joins with subqueries that update the weights as above. By modifying the analyst's query to track and update weights using the DBMS, our CHORUS implementation enables WPINQ to scale to large datasets. We evaluate its performance in Section 7.1.

A.2. The Matrix Mechanism

The matrix mechanism [52] is another general approach for answering a set of counting queries. The insight behind the matrix mechanism is that the optimal way of answering a workload of counting queries might involve first answering a *different* set of queries, then inferring the answers to the workload queries based on these answers. The matrix mechanism is defined in terms of three matrices: the workload queries, represented as a matrix; the *strategy matrix*, which specifies the queries to submit to the database, and a matrix containing the answers to the queries in the strategy matrix. Given these three matrices, the method for answering the workload queries can be specified as a matrix multiplication.

We present an implementation of the matrix mechanism in CHORUS in Figure 8. Its inputs are matrices representing the workload and strategy, and a list of SQL queries corresponding to the strategy queries. We use the Laplace mechanism to answer the strategy queries, then transform the results into a matrix representation and perform the matrix multiplication specified by Li et al. [52] to obtain the workload results. The `mpInverse` method on matrices implements the Moore–Penrose pseudoinverse.

The challenge of determining an optimal set of strategy queries remains; Li et al. [52] consider this an orthogonal problem, and provide some heuristics for developing good strategies. Each of these heuristics can be implemented as Scala functions to generate sets of strategy queries for our implementation of the matrix mechanism.

A.3. Multiplicative Weights (MWEM)

The MWEM algorithm [41] is an iterative algorithm for answering a workload of counting queries with differential privacy. It provides a general algorithmic framework for iteratively improving a differentially private synthetic representation of the underlying data, until the synthetic representation is able to answer the queries in the workload with high accuracy.

Here, we develop an implementation of MWEM for 1-dimensional range queries over a single database table, based on a histogram representation of the data in the table. The full implementation appears in Figure 9. To use the mechanism, the analyst provides a list of range

```
def mwem(queries: List[(Double, Double)],
         bins: List[Double],
         numIters: Int,
         epsilon: Double): List[(Double, Int)] = {
  // answer range query using synthetic representation
  def rangeQuerySyn(synRep: List[(Double, Int)],
                    lower: Double, upper: Double) = {
    var count = 0
    for (i < 0 to synRep.length()) {
      if (i <= lower && i < upper)
        count = count + synRep(i)
    }

    count
  }

  // answer range query using actual data
  def rangeQuery(lower: Double, upper: Double) =
    Select Count(*) From T
      Where C <= lower And C < upper

  // update rule for MWEM
  def mwemUpdate(lower: Double, upper: Double,
                 synRep: MutableList[(Double, Int)],
                 epsilon: Double) = {
    val realAnswer = DB.execute(rangeQuery(lower, upper))
                     + Util.Laplace(1 / epsilon)
    val synAnswer = rangeQuerySyn(synRep, lower, upper)

    val total = synRep.map(_._2).sum()
    for (i < 0 to synRep.length()) {
      if (i <= lower && i < upper)
        synRep(i) = synRep(i) *
         exp((realAnswer   synAnswer) / (2 * total))
    }
  }

  // initialize all counts to 100
  val synRep = bins.map(l => (l, 100))

  // split the privacy budget
  val epsilon_i = epsilon / numIters

  for (i < 0 to numIters) {
    // pick the "worst" query in terms of accuracy
    val bQs = queries.map { case (l, u) =>
      rangeQuery(l, u)   rangeQuerySyn(synRep, l, u) }
    val qIdx = noisyMax(bQs, epsilon_i / 2)

    // update synthetic rep using selected query
    val l, u = queries(qIdx)
    mwemUpdate(l, u, synRep, epsilon_i / 2)
  }
}
```

Figure 9: The MWEM Algorithm in CHORUS

queries (the workload), plus a list of "bin edges" which partition the domain of the table into histogram bins.

At each iteration of the algorithm, we perform two steps: (1) using the exponential mechanism, select a query from the workload which the synthetic representation *cannot* answer with high accuracy; (2) using the Laplace mechanism, obtain a differentially private answer to this query, and use the *multiplicative weights update rule* to update the synthetic representation. The mechanism returns the final synthetic representation.

To simplify the implementation we present here, we require the analyst to specify queries in terms of an upper and lower bound on the desired range. A query of the form (l, u) on column `c` of table `T` is equivalent to the SQL query `SELECT COUNT(*) FROM T WHERE l <= c AND c <= u`.