Architecting a Differentially Private SQL Engine

Ios Kotsogiannis
Duke University

Yuchao Tao
Duke University

Ashwin Machanavajjhala
Duke University

Gerome Miklau
UMass, Amherst

Michael Hay
Colgate University

ABSTRACT

In recent years, *differential privacy* (DP) has emerged as the state-of-the-art for privately analyzing sensitive data. Despite its wide acceptance in the academic community and much work on differentially private algorithm design, there is surprisingly little work on building database systems that allow differentially private query answering using high level, declarative languages like SQL. The lack of such systems has limited the adoption of differential privacy in real-world applications.

In this paper, we propose PRIVSQL, a system architecture for supporting SQL query answering under differential privacy and identify a set of components that can be independently optimized. While there is a mature class of solutions for some components, there is little or no work for others. Our preliminary implementation can support a richer class of SQL queries than a state of the art competitor, with accuracy that is as much as $7000\times$ better.

1. INTRODUCTION

Several organizations want to open the sensitive user data they collect for analysis to their employees and external third parties. For instance, Facebook recently announced a new initiative to allow social scientists to analyze their user data for research into the effect of social media on elections and more generally on democracy [2]. New privacy legislation in the EU and California heavily regulates the analysis and dissemination of user behavioral data, which includes all of their online activity. Thus, there is a need for systems that allow the analysis of sensitive data with strong privacy guarantees. Differential privacy (DP) [5] has arisen as a gold standard for private analysis. DP algorithms are seeing adoption in federal agencies like the US Census Bureau [9] for publishing statistics, in companies like Uber [7] for enabling a private query interface over user data for employees, and in Google Chrome and Apple applications for analyzing user data.

Despite the academic success and growing adoption of differential privacy, it is still extremely hard to analyze data accurately in this model. In fact, each of the deployments mentioned above has required a team of differential privacy experts to design algorithms and tune their parameters.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2019.
9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

There are several challenges to implementing DP algorithms. First, it is difficult to design an algorithm that, given a fixed privacy budget, extracts the most accuracy for a task. A benchmark study [6] evaluated a variety of DP algorithms (for the task of answering range queries) and showed that general purpose algorithms like the Laplace mechanism [5] introduce too much error and that there was no single dominant algorithm for the task. Second, it is easy for implementations to violate differential privacy. There are many examples of mistaken proofs, incorrect implementations, and side-channel attacks that can unravel the protection of a DP algorithm. There is a growing line of work on privacy oriented programming frameworks[11] and a few that focus on accuracy [15] that start to address these issues. However, none of these frameworks has the capabilities of a relational database. There is no support for declarative query answering; an analyst *has* to write a DP program themselves. And most systems only support queries on a single table and none consider updates to the database.

Our vision is to build a differentially private relational database that (a) supports realistic relational schemas containing multiple tables, (b) accurately answers declaratively specified aggregate queries involving standard SQL operators like JOINS, GROUPBY and correlated subqueries, (c) ensures differential privacy with a fixed privacy budget (regardless of how many queries are posed to the system), and (d) allows for database updates.

The above vision reflects our desire to provide interfaces to users (e.g. SQL) that are similar to conventional databases. But the internals of a system for privately answering general SQL queries require components that are unheard of in a standard relational systems. First, above all else, privacy loss must be correctly measured and accounted for. Architecturally this requires (i) a privacy firewall separating the private database from the differentially private outputs and (ii) system components that can statically analyze queries, views, and data transformations, so that any execution in the system can be shown to meet correct privacy loss bounds. Second, in a private system there are no correct query answers, only better or worse approximations of the true query answer. Extracting maximal accuracy for a given bound on privacy loss means (i) we need more input from the user than in conventional systems (their desired 'workload') so the system can adapt to what accuracy means for them; (ii) we need to optimize what we use the privacy budget for (which queries or measurements) and how the privacy budget is divided; and (iii) previously computed results may need to be reused to answer new queries (since they have already been "paid" for). We must shield the user from all this complexity.

Our primary goal in this paper is to articulate a forward-looking system architecture, called PRIVSQL, that is general enough to accommodate new solutions to the challenging sub-problems underlying our architectural components. We present a set of detailed de-

sign principles that any solution to our problem must satisfy (Section 2). We describe our architecture and highlight how different implementations for the components of PRIVSQL trade off accuracy, privacy, and efficiency (Section 3). In Section 4, we describe our ongoing implementation of PRIVSQL and, as evidence of the potential of our architecture, we show that instantiating our components with preliminary component implementations results in dramatic improvements in accuracy over competing approaches, by a factor of as much as $7000\times$.

1.1 Preliminaries

We consider a multi-table relational schema R, containing a set of *sensitive* tables R_s to which privacy protection is guaranteed. We consider differential privacy (DP) [5] as our privacy notion. Informally, the output of a DP algorithm masks the presence or absence of a single tuple in any one of the sensitive tables. More formally, let D be an instance of R. Then $N(D)$ denotes the set of all instances D' that differ in the presence or absence of one tuple in one of the tables in R_s.

Definition 1.1 (Differential Privacy)**.** *A mechanism $\mathcal{M} : R \to \Omega$ is ϵ-differentially private if for any relational database instance D and $D' \in N(D)$, and any set of possible outputs $\forall S \subseteq \Omega$:*

$$Pr[(M)(D) \in S] \leq \exp(\epsilon) Pr[(M)(D') \in S]$$

The privacy guarantee gracefully degrades as multiple DP algorithms execute on the data [5]. The sequential execution of mechanism $M_1, \ldots, M_k$, where M_i satisfies ϵ_i-DP on database instance D is also differentially private with parameter $\epsilon = \sum_i \epsilon_i$.

We consider aggregate SQL queries of the form SELECT COUNT(*) FROM S WHERE Φ, where S is a set of tables and subqueries, and Φ can be an arbitrary predicate possibly containing correlated subqueries. A special subset of aggregate queries are *linear counting queries*, for which S is a single base table, and Φ contains only terminal predicates on the columns of S.

1.2 State of the Art

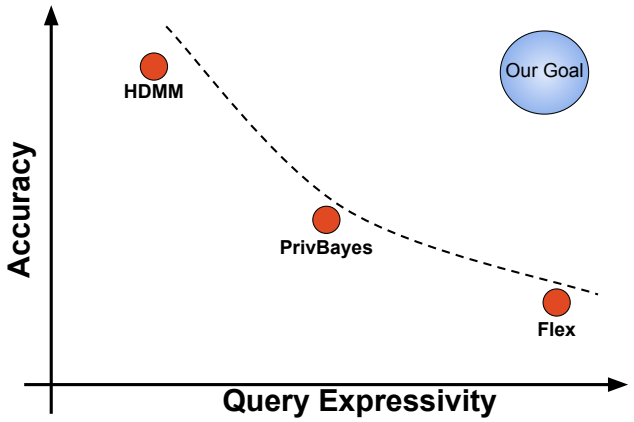

Figure 1: State of the art

We are aware of very few *practical* solutions from the literature that have some of the capabilities mentioned in our vision. One class of solutions takes a set of logically specified linear counting queries *on a single table*, and answers all of them under a fixed privacy budget. The most accurate algorithm in this class is HDMM [10] which represents queries and data as vectors and uses sophisticated optimization and inference techniques to answer them. It can not handle queries with JOINs on single or multiple tables. On the other end of the spectrum is FLEX [7] that can answer a single aggregate SQL query (allowing JOINs but not correlated subqueries) under a given privacy budget. However, unlike HDMM (and similar solutions), (a) the privacy loss incurred by FLEX increases with the number of queries, and (b) FLEX incurs a high utility loss as it answers one query at a time. Finally, another solution is to generate a synthetic database for each base table using PrivBayes [16] under a fixed privacy budget and then use these for query answering. This approach cannot preserve join keys. As shown in Fig. 1, FLEX has the most query expressivity, but limited accuracy; HDMM has the most accuracy, but limited query support; and the solution based on using synthetic data generated by PrivBayes would have medium query expressivity, and as per our experiments (not shown) medium accuracy. Our goal is to build a system that can achieve high accuracy, support high query expressivity, and ensure strong guarantees of differential privacy. Our preliminary implementation supports a richer class of SQL queries than FLEX and offers as much as $7000\times$ better accuracy.

2. PRINCIPLES & JUSTIFICATION

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

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

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

On the other hand, if a private synopsis of the database was constructed with all the privacy budget ϵ_B, then the system could answer any number of queries without further privacy loss as long as they can be answered from the private synopsis. This does not mean than an unlimited number of queries can be *accurately* answered from a private synopsis; see Principle 2 for further discussion.

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

empty intermediate table with the edge table, and hence would return quickly. On the other hand, with Bob in the database, the join may take perceptibly more time, thus revealing the presence of Bob.

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

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

Since we propose to generate a private synopsis, which is already differentially private, (a) no further noise needs to be added and (b) we can ensure that the private synopsis is consistent. A downside of answering queries on a private synopsis is that updates to the database are not reflected in the query answers. We discuss this in more detail in Principle 4.

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

The celebrated result by Dinur-Nissim [4], the *Fundamental Law of Information Reconstruction*, shows that a database containing n bits can be accurately reconstructed by an adversary that submits $n \log^2 n$ counting queries, even if each of the queries has $o(\sqrt{n})$ additive noise. This implies that we cannot hope to accurately answer too large a set of queries from any single synopsis under strong privacy guarantees. It therefore means that we must specify as input a *representative workload* of queries to be answered. This workload can be either a list of explicitly defined queries, or a set of parameterized queries – where constants are replaced by wildcards. The private synopsis will be designed to provide answers to the representative workload with high accuracy. Of course, if the workload contains too many queries then we can not answer all of them with high accuracy without violating the Fundamental Law of Reconstruction. Thus our accuracy guarantees on the queries in the representative workload are best-effort. Our system also tries to answer queries that are not in the input workload and if it can't, then it informs the user.

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

Prior work has shown that queries involving the join of two tables cannot be answered accurately just using private synopses that have been generated *independently* from each of the tables. For instance, Mironov et al. [12] show a $\Omega(\sqrt{n})$ lower bound on the error of computing the intersection between two tables given differentially private access to the individual tables (and not their join). The intuition behind this result follows from the definition of differential privacy. Since join keys are typically unique, no differentially private algorithm can preserve the key. Thus, joins have to be done on coarser quasi-identifiers which are associated with a sufficiently large number of tuples.

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

Principle 4. *Update the private synopsis only if (a) the database has changed significantly (due to updates), or (b) queries posed by the analyst are very different from the stated workload.*

The private synopses generated on a snapshot of the database and an input workload may become out of date either because of database updates, or because the input workload does not capture the current set of queries of interest to an analyst. In this case, the system must be able to update its private synopses. This process will necessarily access the database and in some cases it might require additional consumption of the privacy budget.

We note that no ϵ-DP algorithm can distinguish between answers to a single count query that differ by $< \frac{1}{\epsilon}\log(1/\delta)$ with probability $1-\delta$. That is, for $\epsilon = 0.1$, one can't tell apart counts x and $x+13$ with 95% probability. This range increases as the number of queries increases. Thus, updating the private synopsis for every update to the database is unnecessary and a waste of privacy budget.

3. SYSTEM ARCHITECTURE

Motivated by the above principles, we now describe the architecture of our system for differentially private query answering, called PRIVSQL. The architecture is illustrated in Fig. 2.

3.1 Overview

PRIVSQL stores a relational data instance D, conforming to schema R, in a relational database engine. This engine is placed inside a logical *privacy firewall*. The analyst who poses queries on the database is outside the privacy firewall. While system components inside the firewall are trusted to compute on the database, any process that transfers data across the firewall must provably ensure differential privacy. PRIVSQL has three operational phases: *private synopsis generation*, *query answering*, and *synopsis update*.

The synopsis generation phase is an offline phase that takes as input a 4-tuple $(R, D, \mathcal{Q}, \epsilon)$ and outputs a *set of private synopses* $\tilde{\mathcal{S}}$. R and D are the relational schema and instance, respectively. $\mathcal{Q}$ is a representative workload of queries that may be expressed explicitly or using wildcards. ϵ is the differential privacy budget. The synopsis generation phase makes $\tilde{\mathcal{S}}$ available outside the firewall. Hence, a single execution of the private synopsis generator with a privacy budget ϵ must necessarily satisfy ϵ-differential privacy.

The synopsis generation phase is executed by 4 functionally distinct components. Given the input $(R, D, \mathcal{Q}, \epsilon)$, PRIVSQL first uses the *view selector* (VSELECTOR) to select a set of views $\mathcal{V}$ over the base tables. Next, the *stability calculator* (STABCALC) computes the stability of each view (defined below in Section 3.2), which is in turn used by the *budget allocator* (BUDGETALLOC) to allocate privacy budget ϵ_V to each individual view $V \in \mathcal{V}$. Finally, the synopsis generator (PRIVSYNGEN) takes as input the set of views, the set of privacy budgets, and the workload $\mathcal{Q}$ and returns a set of private synopses $\tilde{\mathcal{S}}$, one synopsis $\tilde{S}_V$ per view $V \in \mathcal{V}$.

The query answering phase answers analyst queries using *only* the set of private synopses $\tilde{\mathcal{S}}$ and other public information. The query answering phase does not access the private database and hence is completely outside the privacy firewall. Thus, query answering does not need to ensure differentially private.

The synopsis update phase generates a new set of private synopses based on the current version of the database instance. This phase can optionally take as a input a new representative workload

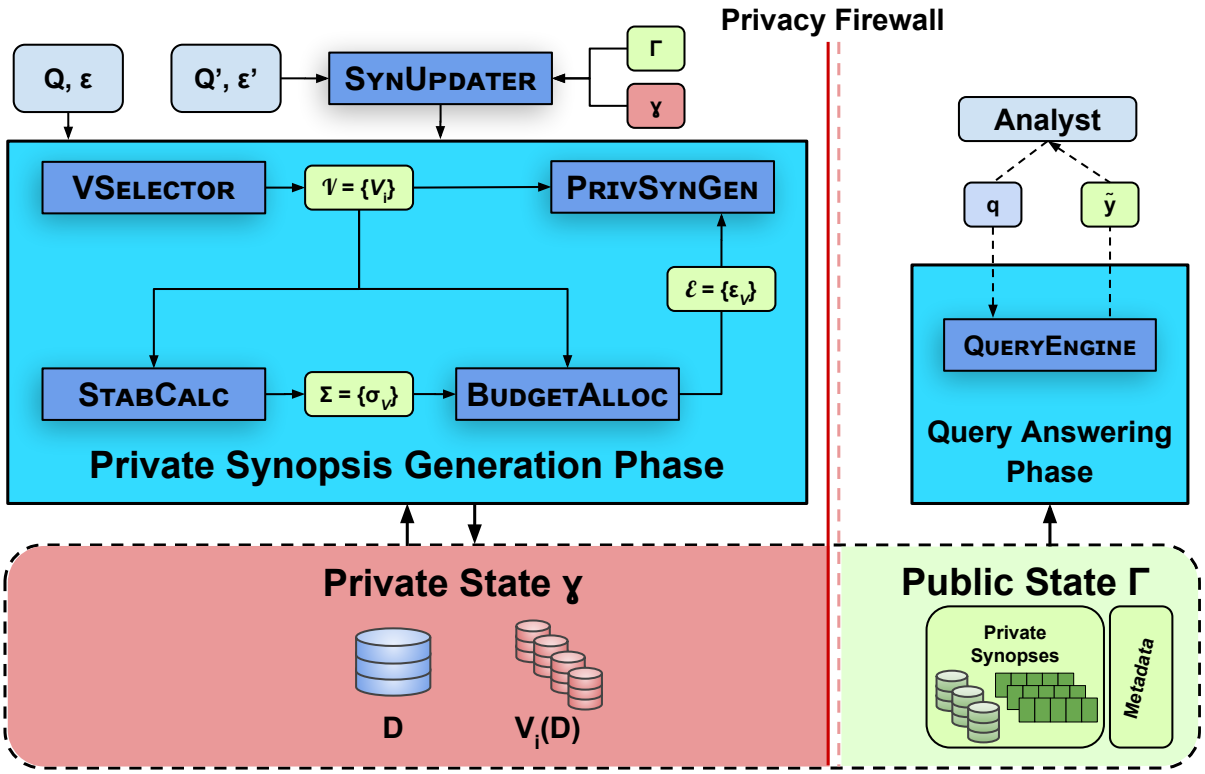

Figure 2: System Architecture

of queries $\mathcal{Q}_{new}$, and an additional privacy budget ϵ_{new}, and reruns the synopsis generation phase. The cumulative privacy loss incurred by the database is at most the sum of the ϵ values associated with each run of the synopsis update phase.

3.2 System Components

System State: All the data maintained by PRIVSQL is stored in the *system state*, which is denoted by the tuple (γ, Γ). γ denotes the *private state* and is within the privacy firewall. γ contains the private database instance D and any data derived from it (like view instances) using algorithms that are not differentially private. The system state allows cross-component communication and isolates the private data from the publicly releasable data.

Γ denotes the *public state* and is outside the privacy firewall. Γ contains public information about the database, like the schema R, any instances in the schema with no privacy concerns (e.g. a dimension hierarchy on a public attribute like geography), outputs of differentially private algorithms (e.g., the private synopses), and other meta-data. Intermediate results of the synopsis generation phase that can be publicly released (like the set of view definitions selected and other examples seen later) are also be stored in Γ.

View Selector: The goal of VSELECTOR is to select a set of view definitions $\mathcal{V}$ over the base relational schema R. These view definitions are made available in the public state Γ for query answering. On the other hand, view instances (i.e., the materialized views), are maintained as part of the private state γ. VSELECTOR may be data independent, in which case it only takes as input the schema R and $\mathcal{Q}$. A *data dependent* VSELECTOR expends a part of the privacy budget and can select views based on the private instance D.

A query q is said to be *answerable* by view definition V if there exists a rewriting q_V such that for every instance D of the schema we have $q(D) = q_V(V(D))$. As per Principle 3, every query $q \in \mathcal{Q}$ should ideally be answerable by one of the views output by VSELECTOR. The output $\mathcal{V}$ of VSELECTOR is $\mathcal{Q}$-lossless if every query $q \in \mathcal{Q}$ is *answerable* in at least one of the views.

Private Synopsis Generator: PRIVSYNGEN takes as input a view instance $V(D)$, a set of queries $\mathcal{Q}_V \subseteq \mathcal{Q}$, and ϵ_V, a privacy budget specific to the view and returns a private synopsis $\tilde{S}_V$ that is made available to the public state Γ. A synopsis can be a set of synthetic tuples, query answers, vectors, histograms, or hierarchical counts. To generate a synopsis, PRIVSYNGEN runs on $V(D)$ an appropriate ϵ_V-differentially private algorithm that is designed to minimize the error on the query workload $\mathcal{Q}_V$.

Stability Analyzer: Executing an ϵ_V-DP algorithm on a view instance $V(D)$ hides the effect of adding or removing a tuple to the *view instance* on the output synopsis $\tilde{S}_V$, but may not offer similar protection to base tables contributing to the view. That is, it does not necessarily ensure ϵ_V-DP with respect to the base tables. For instance, consider a V that joins two base tables. Adding a record t to one of the base tables might result in σ tuples added to $V(D)$ if the join key corresponding to t has multiplicity σ in the other table.

To address this, we must consider the *stability* of a view, which is defined to be the maximum number of tuples σ_V that are added or removed to the view instance due to adding or removing one row in a base table, over all possible database instances D. Executing an ϵ_V-DP algorithm on $V(D)$ can be shown to satisfy $\sigma_V \epsilon_V$ DP over the base tables [11]. The STABCALC component takes as input a view V and outputs its stability σ_V, which is then added to Γ.

Privacy Budget Allocator: The BUDGETALLOC divides the total privacy budget ϵ into a set of privacy budgets $\mathcal{E} = \{\epsilon_{view}\} \cup \{\epsilon_V\}_{\forall V \in \mathcal{V}}$ such that the synopsis generation phase satisfies ϵ-DP. ϵ_{view} is allocated to VSELECTOR, and is 0 if VSELECTOR is data

independent. ϵ_V is the privacy budget used by PRIVSYNGEN to generate $\tilde{S}_V$ on view V. An allocation that satisfies:

$$\epsilon_{view} + \sum_{V \in \mathcal{V}} \sigma_V \epsilon_V \leq \epsilon \quad (1)$$

always guarantees that the synopsis generation phase satisfies ϵ-DP.

To ensure that the synopsis generation phase provably guarantees differential privacy, the privacy sensitive components can be implemented in a system like ϵktelo [15], so that the proof of privacy comes automatically with the implementation .

Query Answering Engine: This is the interface that the analyst has access to outside of the privacy firewall. Given the synopses $\tilde{S}$ as well as the rest of the public state Γ, the QUERYENGINE returns answers to input queries. For input query q, the QUERYENGINE first identifies which view $V \in \mathcal{V}$ contains it and then rewrites q in terms of V – rather than in terms of the base tables. If no view supports q then the QUERYENGINE outputs $\perp$ meaning that this query can not be currently answered. Next, if it finds a view V, the QUERYENGINE uses the appropriate synopsis $\tilde{S}_V$ to answer the query. If there are public tables, QUERYENGINE can directly answer queries answerable by the public tables, and these will have no noise. Note that query answering is outside the privacy firewall and does not require the use of differentially private algorithms.

Synopsis Updater: SYNUPDATER is a novel component that is typically not considered in differentially private query answering. It takes as input the current state of the system (γ, Γ), and optionally a new query workload $\mathcal{Q}_{new}$, and a new privacy budget ϵ_{new}, and decides whether to rerun the synopsis generation phase or not.

3.3 Implementation Considerations

PRIVSQL's goal is to ensure private, accurate and efficient query answering. When the queries are all linear counting queries , there exist differentially private methods that can efficiently answer queries with near-optimal accuracy [10]. However, such a mechanism is not known to exist for sets of complex SQL queries. Jointly optimizing the implementations of all of the components in PRIVSQL is hard, and we conjecture the problem is intractable. In this section we highlight a few alternate implementations of each component, and discuss how they tradeoff privacy, accuracy and efficiency of query answering.

View Selector: As per Principle 3, every query in $\mathcal{Q}$ must be answerable by one of the views output by the view selector. This can be achieved by generating a single denormalized universal view that encodes all joins. However, this is unlikely to be optimal in terms of accuracy or efficiency. The single view will have a high stability if it involves multiple many-to-many joins. Recall that given a fixed privacy budget, high stabilities result in high error. Maintaining a set of views, each encoding a subset of the joins that appear in $\mathcal{Q}$ would result in views with lower stability. Moreover, a single view would be large both in terms of rows and columns slowing down the synopsis generator that operates on this view.

Another design choice is to select one specialized view for every query in $\mathcal{Q}$. This is also undesirable as the number of views may be too large: if $\mathcal{Q}$ contained the set of all range queries on the Age attribute, $\mathcal{Q}$ would have about 100×100 queries, and thus as many views. Having a single view that supports all range queries on an attribute would also be better since this is a task for which there are well-known, accuracy-optimal differentially private algorithms that PRIVSYNGEN can employ.

Private Synopsis Generator: This component is probably the most well understood as it is an instance of a common problem studied in DP literature – answering a set of queries on a single table. Efficient and accurate methods are known both for releasing a synthetic dataset in the original schema of the table [16], as well as releasing vectors of counts [6, 10] that minimize the error for linear counting queries. One consideration is whether to release synthetic tuples or vectors of counts. The former is efficient in terms of representation – the vector form encodes one count for every possible tuple in the cross product of the domains of the attributes in the table, and is thus exponential in the number of attributes. However, the latter allows maintaining fractional counts, which leads to lower error. In addition, vector form allows the use of linear algebra based inference methods to reason across multiple independent noisy releases, which can help answer queries not present in $\mathcal{Q}$.

Stability Calculator: We conjecture that the query complexity of computing the stability of a view is NP-hard, as it is closely related to the problem of query evaluation (which is also hard for queries that include self or cyclic joins). Thus, efficient implementations of STABCALC necessarily need to compute an upper bound on stability. A standard approach [7, 11] computes the stability of a view definition by considering a specific query plan (rather than the actual query), treating it as a sequence of transformations on the data, using rules to estimate the stability of each transformation, and multiplying the individual stabilities to get the stability of the view. However, as we show in Section 4, this approach can overestimate the stability by over three orders of magnitude.

Budget Allocator: Any allocation of the total budget that satisfies Eq. (1) ensures that the private synopsis generation satisfies ϵ-DP. This allocation impacts which workload queries end up answered with greater relative accuracy over others. A *naive* method is to divide ϵ equally and assign $\epsilon' = \epsilon/\sigma_V$ to each view V. Under this naive allocation, views involving joins (with typically larger stabilities) have lower privacy budgets and thus will support query answering with higher errors. A *view-fair* allocation strategy would ensure that for two views V, V', $\epsilon_V = \epsilon_{V'} = \epsilon/\sum_{V \in \mathcal{V}} \sigma_V$. However, one view may support many more queries than another view, so the budget allocation may also take that into account. One could go further and take into account the number of queries supported by a view during allocation. The design of a truly *query fair* budget allocator requires knowledge of the downstream differentially private mechanism used and the query answering engine in addition to the set of queries supported by each view.

Synopsis Updater: A key decision to be made in the implementation of the SYNUPDATER is whether or not to expend additional privacy budget. One can refresh the synopses without incurring additional privacy loss if the new synopses are only generated on rows that were inserted into the base tables *after* the previous synopsis was generated. In that case, the privacy guarantee of the old and new synopsis generation would compose under *parallel composition*. However, this might not be accurate for queries that involve old and new tuples. First, each private synopsis will have noise and adding up query answers across synopses will result in the noise adding up. Second, if some of the views involve joins, then tuples in the view that join old tuples with new tuples will not be present either in the old or the new synopsis.

Another decision is the frequency at which SYNUPDATER is run. Frequent synopses updates allow the query answering engine to use a recent version of the database. As a consequence, the system either spends larger amounts of privacy budget, or ends up creating many synopses on smaller subsets of the data (when each run of SYNUPDATER uses $\epsilon_{new} = 0$). Both cases lead to high error in query answers. On the other hand, running the SYNUPDATER less

frequently means the queries are answered on a more out-of-date version of the database, but with higher accuracy.

One way to implement the SYNUPDATER is to adapt recent techniques [3] that allow rerunning DP algorithms on unboundedly growing databases with small additional privacy loss.

Query Engine: The implementation of QUERYENGINE and the queries it can support are closely tied to the form of the private synopses $\tilde{\mathcal{S}}$. If the synopsis for a view is a relation in tabular form, then the QUERYENGINE can process and answer any query that is answerable by that view. However, only those queries that were in the input $\mathcal{Q}_V$ would have low error. On the other hand, if the synopsis is a vector of counts, then the QUERYENGINE would have to not only rewrite the input query q as a query q_V on the view, but also represent it as a linear counting query over the synopsis. In this case, the QUERYENGINE could only support SELECT COUNT(*) queries with low error. QUERYENGINE could rewrite non-count queries (e.g. median, top-k, GROUP BY ... HAVING) as postprocessings over another set of linear queries. For instance, the median can be computed from a histogram or a CDF, both of which are sets of counting queries. However, as discussed before (and as we will see in Section 4), existing algorithms that generate synopses in tabular form typically introduce high error during query answering.

4. IMPLEMENTATION & RESULTS

We now briefly describe our ongoing implementation of PRIVSQL components, and present preliminary results.

View Selector: We use a data-independent VSELECTOR that groups together queries with the same join structure under the same view and makes each view as small as possible while still enjoying the $\mathcal{Q}$-lossless property.

VSELECTOR iteratively transforms each query of $\mathcal{Q}$ to a query-view pair $(\bar{q}, V)$ such that $\bar{q}$ is a linear counting query on V and the transformation is equivalent, i.e., for all databases D: $\bar{q}(V(D)) = q(D)$. First, q is decorrelated [13], i.e., the correlated subqueries in q are transformed into joins. Now q is of the form SELECT COUNT(*) FROM $T_1, \ldots, T_k$ WHERE Φ_{join} AND Φ. Here T_i are base tables or subqueries, Φ_{join} are join conditions and Φ contains no subqueries. V is set to SELECT * FROM $T_1, \ldots, T_k$ WHERE Φ_{join}, and $\bar{q}$ is set to SELECT COUNT(*) FROM V WHERE Φ. Transformed queries corresponding to the same view V form $\mathcal{Q}_V$, the workload for view V. We project out columns from V that are not mentioned in any $\bar{q} \in \mathcal{Q}_V$

Synopsis Generator: Let $dom(V)$ be the domain of a view V, i.e., the cross product of the domains of all attributes in V. Then, for a materialized view $V(D)$, PRIVSYNGEN either releases synthetic tuples, or a vector of counts for that view. This decision is made based on $|dom(V)|$. When $dom(V) < 10^6$, PRIVSYSGEN represents $V(D)$ as a vector $\mathbf{x}_V$, and the workload $\mathcal{Q}_V$ as a matrix W_V. $\mathbf{x}_V$ is a histogram of counts over the cross product of all the attributes in V, and W_V is such that query answers are preserved (i.e., $\mathcal{Q}_V(V(D)) = W_V \cdot \mathbf{x}_V$). This representation permits the use of standard vector based techniques (e.g., from ϵktelo [15]) like IDENTITY (which adds noise to each entry in $\mathbf{x}_V$), WORKLOAD (which adds appropriately scaled noise to $W_V \cdot \mathbf{x}_V$), and DAWA [8]. For larger domain sizes $dom(V) \geq 10^6$, vector based approaches are too slow, and PRIVSYSGEN uses PRIVBAYES [16] to produces synthetic tuples.

Stability Calculator: Our STABCALC implementation builds on elastic sensitivity (ES) (which is used in FLEX [7]), the state of the art method for estimating stability of views. It works as explained in Section 3.3 by analyzing a specific query plan for the view and recursively computes the stability starting from the leaves of the query tree to the root. We extend ES's rules for computing stability in a number of ways, but due to space limits we only outline one of the ideas that has the most impact on final error.

In addition to the view definition, ES requires as input the maximum multiplicity of any value in each attribute in the base tables to determine the effect of adding a tuple on the output of a join.[1] ES derives the maximum multiplicity of a value in a derived intermediate table based on worst case assumptions about how tuples might join. We recognize that often the maximum multiplicity of values of certain attributes in derived tables can be accurately ascertained based on the query. For instance, the maximum frequency of any value in the 'A' column in the output of a SELECT A, ... FROM T GROUP BY A is 1, since A is now a primary key. We observe this subquery appears in a number of real world queries (including the ones we use in our experiments).

Table 1: L2 error of PRIVSQL and FLEX for 3 datasets.

		Datasets		
		Block	Puma	NC State
System	FLEX	453,892	453,892	453,892
	PRIVSQL$_I$	307.25	307.25	307.25
	PRIVSQL$_D$	129.37	171.00	362.57
	PRIVSQL$_W$	62.86	62.86	62.86

Budget Allocator: We use naive allocation (see Section 3.3).

Query Engine: For an input query q, QUERYENGINE first uses the VSELECTOR's query transformation to construct V and the linear query $\bar{q}$. If V is not one of the materialized views, QUERYENGINE returns $\perp$. Else, if the synopsis $\tilde{S}_V$ is a table, it simply returns $\bar{q}(\tilde{S}_V)$. If $\tilde{S}_V$ is a vector, it transforms $\bar{q}$ also into a 0/1 vector returns its dot product with the data vector.

SynUpdater: If $\epsilon' > 0$, we rerun the synopsis generation phase with input $(R, D', \mathcal{Q}', \epsilon')$ and create additional synopses.

4.1 Preliminary Results

Setup: We consider an analyst submitting queries to a private database of people in the US (we use a publicly available Census dataset [14]). The database schema has 2 tables: PERSONS(ID, SEX, GENDER, AGE, RACE, HID), HOUSING(HID, LOCATION). We construct 3 versions of the datasets by considering only rows corresponding to people living in a specific census block, a specific PUMA (a Census region) and the state of NC. The three cases result in a PERSON table with $1K$, $50K$, or $> 5.4M$ tuples.

We constructed a workload $\mathcal{Q}'$ of 192 queries by parsing the descriptions of tables released by the US Census Bureau as part of the Summary File 1 (SF-1) [1]. Since the US Census Bureau releases these tables, we assume these are queries that users of census data consider important. We chose a subset of 65 queries $\mathcal{Q} \subset \mathcal{Q}'$ as the representative queries input to the synopsis generator. We used QUERYENGINE to answer all queries in $\mathcal{Q}'$.

We use a total privacy budget $\epsilon = 1$. We compare against FLEX [7], and use it to answer each query in $\mathcal{Q}'$ with a budget of $\epsilon/192$.

Results: Based on $\mathcal{Q}$, VSELECTOR chose 17 views to materialize including the base table and various joins between the PERSON and HOUSING table. Each of these views had $dom(V) < 10^6$, so PRIVSYSGEN did not use PrivBayes to generate the synopsis.

[1] To strictly satisfy ϵ-DP, one should not use properties of D. We have methods to estimate the maximum multiplicities using a portion of the privacy budget. To simplify presentation, we assume maximum multiplicities of attributes are publicly known for our implementation and for ES/FLEX.

In Table 1 we see our results for 3 instantiations of our private engine depending on the PRIVSYSGEN module – PRIVSQL$_W$ uses WORKLOAD to construct the private synopsis while PRIVSQL$_I$ uses IDENTITY and PRIVSQL$_D$ uses DAWA. We execute each system 10 times and report the average L2 per query error: $\sqrt{\sum_{q \in \mathcal{Q}'}(q(D) - \tilde{q}(D))^2}/|\mathcal{Q}'|$, where $\tilde{q}(D)$ is the noisy answer.

We see that all instantiations of PRIVSQL outperform FLEX by at least 3 orders of magnitude. Moreover, we can see that for this problem PRIVSQL$_W$ is the most suitable and overall it offers an $7,220\times$ improvement. If STABCALC had used ES (as in FLEX), the improvement would have been $13\times$. Our improvements to stability calculation contribute to a $550\times$ improvement.

5. CONCLUSIONS

We presented an architecture, called PRIVSQL, for building a first of a kind differentially private relational database engine that can (a) handle relational schemas with multiple tables, (b) answer declaratively specified aggregate queries involving standard SQL operators like JOINS, GROUPBY and correlated subqueries with high accuracy on a prespecified workload of queries, (c) ensure differential privacy with a constant privacy budget (no matter how many queries are posed to the system), and (d) allow for updates to the underlying database. Our preliminary implementation can support a richer class of SQL queries than competing approaches while offering dramatic improvements in accuracy.

Our system architecture is modular and each component can be improved independently to improve the accuracy, privacy and efficiency of the system. In fact, innovations in some of these components do not require a deep knowledge of differential privacy, and many problems are in fact very relevant to the database community. For instance, view selection is a classic problem in databases, and the hardness of computing stability of a view is closely related to the hardness of query evaluation. Query answering using inference techniques over approximate, incomplete and noisy views is a topic of interest both in the privacy community as well as in online query exploration and approximate query processing. We hope our architecture has clarified the challenges in designing DP systems to the database community.

6. REFERENCES

[1] 2010 census summary file 1. https://www.census.gov/prod/cen2010/doc/sf1.pdf.

[2] A new model for industry-academic partnerships. https://gking.harvard.edu/partnerships, 2018 Working Paper.

[3] R. Cummings, S. Krehbiel, K. A. Lai, and U. Tantipongpipat. Differential privacy for growing databases. *CoRR*, abs/1803.06416, 2018.

[4] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *ACM PODS*, 2003.

[5] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 2014.

[6] M. Hay, A. Machanavajjhala, G. Miklau, Y. Chen, and D. Zhang. Principled evaluation of differentially private algorithms using dpbench. In *ACM SIGMOD*, 2016.

[7] N. Johnson, J. Near, and D. Song. Practical differential privacy for SQL queries using elastic sensitivity. *PVLDB*, 11(5), 2018.

[8] C. Li, M. Hay, G. Miklau, and Y. Wang. A Data- and Workload-Aware Algorithm for Range Queries Under Differential Privacy. *PVLDB*, 7(5), 2014.

[9] A. Machanavajjhala, D. Kifer, J. M. Abowd, J. Gehrke, and L. Vilhuber. Privacy: Theory meets practice on the map. In *ICDE*, 2008.

[10] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *PVLDB*, 11(10), 2018.

[11] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *ACM SIGMOD*, 2009.

[12] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Advances in Cryptology - CRYPTO 2009*.

[13] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE*, 1996.

[14] W. Sexton, J. M. Abowd, I. M. Schmutte, and L. Vilhuber. Synthetic population housing and person records for the united states. https://doi.org/10.3886/E100274V1.

[15] D. Zhang, R. McKenna, I. Kotsogiannis, G. Miklau, M. Hay, and A. Machanavajjhala. ϵktelo: A framework for defining differentially-private computations. In *ACM SIGMOD*, 2018.

[16] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. Privbayes: Private data release via bayesian networks. In *ACM SIGMOD*, 2014.