PrivateSQL: A Differentially Private SQL Query Engine

Ios Kotsogiannis[‡], Yuchao Tao[‡], Xi He[⊕], Maryam Fanaeepour[‡]
Ashwin Machanavajjhala[‡], Michael Hay[†], Gerome Miklau[⋆]
[‡] Duke University, [⊕] University of Waterloo, [†] Colgate University, [⋆] University of Massachusetts, Amherst
{iosk, yctao, maryam, ashwin}@cs.duke.edu, xihe@uwaterloo.ca, mhay@colgate.edu, miklau@cs.umass.edu

ABSTRACT

Differential privacy is considered a de facto standard for private data analysis. However, the definition and much of the supporting literature applies to flat tables. While there exist variants of the definition and specialized algorithms for specific types of relational data (e.g. graphs), there isn't a general privacy definition for multi-relational schemas with constraints, and no system that permits accurate differentially private answering of SQL queries while imposing a fixed privacy budget across all queries posed by the analyst.

This work presents PrivateSQL, a first-of-its-kind end-to-end differentially private relational database system. PrivateSQL allows an analyst to query data stored in a standard database management system using a rich class of SQL counting queries. PrivateSQL adopts a novel generalization of differential privacy to multi-relational data that takes into account constraints in the schema like foreign keys, and allows the data owner to flexibly specify entities in the schema that need privacy. PrivateSQL ensures a fixed privacy loss across all the queries posed by the analyst by answering queries on private synopses generated from several views over the base relation that are tuned to have low error on a representative query workload. We experimentally evaluate PrivateSQL on a real-world dataset and a workload of more than $3,600$ queries. We show that for 50% of the queries PrivateSQL offers at least $1,000\times$ better error rates than solutions adapted from prior work.

PVLDB Reference Format:
Ios Kotsogiannis, Yuchao Tao, He Xi, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, Gerome Miklau. PrivateSQL: A Differentially Private SQL Query Engine. *PVLDB*, 12(11): 1371-1384, 2019.
DOI: https://doi.org/10.14778/3342263.3342274

1. INTRODUCTION

Differential privacy is widely accepted in academia as the gold standard for private data analysis. An algorithm is differentially private if its output does not change significantly due to input changes. This ensures privacy when changes in the input correspond to adding or removing an individual's data and privacy is quantified by a parameter ϵ, called the *privacy loss budget*. Differential privacy is typically achieved by carefully injecting noise into the outputs. Recently, we have seen several real-world deployments at Google [8, 16], Apple [11], and the US Census Bureau [2, 19, 30].

Despite the academic success and growing adoption, it is still extremely hard for non-experts to use differential privacy. In particular, it is difficult to correctly define the privacy semantics as well as to design an algorithm which, given a fixed privacy budget and privacy semantics, offers the greatest accuracy for a task. Hence, each of the aforementioned deployments has required a team of privacy experts to design accurate algorithms that satisfy the privacy definition appropriate for the data.

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

Second, there are no known algorithms for accurately answering complex queries over relational databases involving JOINS, GROUPBY and correlated subqueries. Algorithms are known for accurately answering special classes of queries like statistical queries (e.g., histograms, CDFs, marginals on a single table) [7, 20, 34, 35, 37, 40], sub-graph queries (e.g., triangle counting, degree distribution) [10, 12, 21, 25, 26], and monotone queries (e.g., counts on joins) [9]. The closest competitor to our work in terms of query expressivity is Flex [24], which only offers support for specific and limited privacy semantics that do not necessarily translate to real-world policies. Flex does not support queries that have correlated subqueries or subqueries with GROUPBY operations (e.g. it cannot support degree distribution queries).

Third, there are no known algorithms for accurately answering sets of complex queries under a common privacy budget. Sophisticated algorithms are known for answering sets of statistical queries on a single table [27]. Such mechanisms do not exist for graphs and SQL queries, and all prior work only optimizes error for single queries.

Our vision is to lower the barrier to entry for non-experts by building a differentially private relational database that

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342274

(a) supports privacy policies on realistic relational schemas with multiple tables, (b) allows analysts to declaratively query the database via aggregate queries involving standard SQL operators like JOINS, GROUPBY and correlated subqueries, (c) automatically designs a strategy with low error tuned to the privacy policy and analyst queries, and (d) ensures differential privacy with a fixed privacy budget over all queries posed to the system. While there is a growing line of work on privacy oriented programming frameworks [32, 38] that share the goal of helping non-experts, none of these support relational data and declarative query answering; analysts *must* write differentially private programs themselves.

Our contributions are as follows:

- PrivateSQL is a first of its kind end-to-end differentially private relational database system. PrivateSQL exposes a differentially private SQL query answering interface to analysts. PrivateSQL accurately answers SQL queries while imposing a fixed privacy budget across all queries posed by the analyst.
- PrivateSQL allows privacy to be specified at multiple resolutions using a novel generalization of differential privacy for multi-relational databases with constraints. Our generalization captures popular variants of differential privacy that apply to specialized examples of relational data (like Node- and Edge-DP for graphs).
- PrivateSQL employs a new methodology for answering complex SQL counting queries under a fixed privacy budget. Our algorithm identifies a set of views over base relations that support common analyst queries and then generates differentially private synopses from each view. Queries posed to the database are rewritten as linear counting queries over a view and answered using only the private synopsis corresponding to that view, resulting in no additional privacy loss.
- Using a variety of novel techniques like policy-aware rewriting, truncation, and constraint-oblivious sensitivity analysis, PrivateSQL ensures that the private synopses generated from views provably ensure privacy as per the data owner's privacy policy, and have high accuracy.
- We evaluate PrivateSQL on use cases inspired by the U.S. Census data releases and the TPC-H benchmark. On a workload of >3,600 real world SQL counting queries inspired by the Census and $\epsilon = 1$, 50% of our queries incurred $< 6\%$ relative error. In comparison, a system that uses the state-of-the-art Flex [24] incurs $> 100\%$ error for over 65% of the queries; i.e., Flex has worse error for these queries than a trivial baseline method that returns 0 for every answer (see Fig. 9b).

2. OVERVIEW

PrivateSQL is designed to meet three central goals:

- *Workloads*: The system should answer a workload of queries with bounded privacy loss.
- *Complex Queries*: Each query in the workload can be a complex SQL expression over multiple relations.
- *Multi-resolution Privacy*: The system should allow the data owner to specify which entities in the database require protection.

In this section, we outline key ideas that enable PrivateSQL to support these goals and describe the system architecture.

2.1 Key Ideas

Prior work [24] has proposed differentially private techniques for answering a single query given a fixed privacy budget. This does not naturally extend to workload answering as the privacy loss compounds for each new query answered. Further, by the "fundamental law of information reconstruction" [13] running such a system indefinitely would leak enough information to rebuild the entire database.

Workloads answered using synopses: To support a workload of queries, our first key idea is to construct *synopses*. A synopsis captures important statistical information about the database – analogous to pre-computed samples in approximate query processing [4]. The privacy loss budget is spent constructing and releasing the synopses. Once released, subsequent queries are answered using only a synopsis and not the private database. As the synopsis is public, there is no privacy cost to querying it and an unlimited number of queries can be answered (though the fundamental law also implies that some queries will be poorly approximated).

Synopses generated for selected views: There is considerable prior work on generating a differentially private statistical summary for a single table. Such strategies have been shown to support workloads of simple (linear) queries. But if a synopsis were generated for each base table in the schema, it is known that complex queries, such as the *join* of two tables, would be poorly approximated [33].

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

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

This brings us to the third key idea: we introduce novel techniques for calculating bounds on view sensitivity. Exact sensitivity calculation is hard, even undecidable [5]. We employ a rule-based calculator to each relational operator in the view definition (which is expressed as a relational algebra expression). The per operator bounds compose into an upper bound on the true sensitivity of the entire view. An additional challenge is that some queries have high, even unbounded, sensitivity because of worst case inputs. The

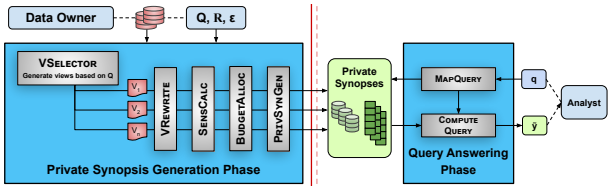


Figure 1: Architecture of the PrivateSQL System

previous example has a sensitivity that is equal to the size of the largest possible household. We address high sensitivity queries by *truncation*, an operation that drops records that cause high sensitivity (e.g., large households). Truncation significantly lowers the variance in query answers at the expense of introducing bias due to data deletion. We describe techniques to *privately* estimate truncation thresholds and we empirically explore the bias-variance tradeoff.

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

Our fourth key idea is extending differential privacy to the multi-relational setting. With our approach, one relation is designated as the *primary private relation*, but the privacy protection extends to other *secondary private* relations that refer to the primary one through foreign keys. We show this allows the data owner to vary the privacy resolution (e.g., to choose between protecting an individual vs. an entire household and all its members). We describe this extension in Section 4 and relate it to prior literature.

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

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

2.2 System Architecture

We briefly review the architecture of PrivateSQL (illustrated in Fig. 1) and the algorithms for the two main operational phases. The first phase is *synopsis generation* where a representative workload is used to guide the selection of views followed by the differentially private construction and publication of a synopsis for a chosen set of views. The second phase is *query answering* where each user query is mapped to the appropriate view and then answered using the released synopsis of that view.

Algorithm 1 Synopsis-Generation

Input: Schema $\mathbb{S}$, private database instance $\mathbf{D}$, representative workload $\mathcal{Q}$, privacy policy $P = (\mathtt{R}, \epsilon)$.
Output: A set of views $\mathcal{V}$ and private synopses $\{\tilde{S}_V\}_{V \in \mathcal{V}}$

1: $\mathcal{V} \leftarrow$ VSelector$(\mathbb{S}, \mathcal{Q})$ ▷ Choose views based on workload
2: Reserve ϵ_{mf} to estimate thresholds for relations in views.
3: $\epsilon \leftarrow \epsilon - \epsilon_{mf}$
4: **for** each view V in $\mathcal{V}$ **do**
5: $\quad V^{\tau,\ominus} \leftarrow$ VRewriter$(V, P, \mathbb{S})$
6: $\quad \tau_V \leftarrow$ Estimate truncation thresholds using $\epsilon_{mf}/|\mathcal{V}|$
7: $\quad \hat{\Delta}_V \leftarrow$ SensCalc$(V^{\tau,\ominus}, \mathbb{S}, \tau_V)$
8: $\quad Q_V \leftarrow \{\bar{q} \mid q \in \mathcal{Q} \wedge$ MapQueryToView$(q, \mathbb{S}) = (\bar{q}, V)\}$
9: **for** each $V \in \mathcal{V}$ **do**
10: $\quad \epsilon_V \leftarrow$ BudgetAlloc$(V, [Q_V], [\hat{\Delta}_V], \epsilon)$
11: $\quad \tilde{S}_V \leftarrow$ PrivSynGen$(V^{\tau,\ominus}, V^{\tau,\ominus}(\mathbf{D}), \epsilon_V, Q_V)$
12: **return** $(V, \tilde{S}_V)$ for each $V \in \mathcal{V}$

Synopsis generation (described in Algorithm 1) takes as input a database instance $\mathbf{D}$, which is private, and its schema $\mathbb{S}$, which is considered public. It also takes a representative query workload of SQL queries, $\mathcal{Q}$, and a privacy policy $P = (\mathtt{R}, \epsilon)$ that specifies a privacy budget ϵ and a *primary private relation* $\mathtt{R}$ (formally defined in Section 4).

First, the VSelector module (line 1) uses the representative workload $\mathcal{Q}$ to select a set of view definitions $\mathcal{V}$. Each view (interpreted as a relational algebra expression) is rewritten using the VRewriter module (line 5) in two ways. First, *truncation operators* are included when there is a join on at attribute that may result in an unbounded number of output tuples. Truncation enforces a bound on the join size by dropping join keys with a multiplicity greater than a threshold. The thresholds are learned from the data (line 6) in a differentially private manner. Next, base tables in the view definition are rewritten using *semijoin operators*, making explicit the foreign key dependencies between the primary private relation and other tables. This ensure that the computed sensitivity matches the privacy policy.

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

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

We now generate a synopsis for each view V. Each synopsis is allocated a portion of the total privacy loss budget. The BudgetAlloc component (line 10) determines the allocation based on factors like view sensitivity and/or the size of Q_V. Finally, the PrivSynGen component takes as input the view definition, view instance $V(\mathbf{D})$, a set of linear queries Q_V, and a privacy budget ϵ_V and returns a differentially private private synopsis $\tilde{S}_V$. This module runs an ϵ_V-differentially private algorithm and outputs either a set of sythetic tuples or a set of query answers (like histograms or a set of counts).

We present our generalization of differential privacy for relational databases in Section 4. We outline VSelector

in Section 5. We describe SensCalc and the truncation rewrite in Section 6, and the semijoin rewrites in Section 7. PrivSynGen and BudgetAlloc are described in Section 8.

Query answering using views is a well studied problem [18] and in PrivateSQL is performed by the query answering phase. More specifically, it uses the function MapQueryToView, described above, to convert q into a query $\bar{q}$ that is linear on a view V. If V is one of the views for which PrivateSQL generated a synopsis, then $\bar{q}$ is then executed on the appropriate private synopsis to produce an answer. If the query cannot be mapped to any view, it returns $\perp$.

3. NOTATION

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

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

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

Definition 3.1. *Given relations R, S and primary key A_{pk} in R, a foreign key can be defined as:*

$$S.A_{fk} \rightarrow R.A_{pk} \equiv S \ltimes_{A_{fk} A_{pk}} R = S$$

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

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

SQL queries supported: In Fig. 2 we present the grammar of PrivateSQL supported queries. We consider aggregate SQL queries of the form SELECT COUNT(*) FROM S WHERE Φ, where S is a set of relations and sub-queries, and Φ can be a positive boolean formula (conjunctions and disjunctions, but no negation) over predicates involving attributes in S. We support equijoins and subqueries in the WHERE clause, which can be correlated to attributes in the outer query. The grammar does not support negations, non-equi joins, and joins on derived attributes as tracking sensitivity becomes a challenging and even intractable [5] for such queries. Extensions for sum/median query support are discussed in Section 10.

AggQuery ::= SELECT COUNT(*) FROM TableList

TableList ::= Table | Table, TableList

Table ::= R | SELECT [AttrList,] [COUNT(*)] FROM TableList [WHERE Exp] [GROUPBY AttrList]

AttrList ::= A | A, AttrList

Exp ::= Literal | Exp AND Exp | Exp OR Exp

Literal ::= A op A | A op *val* | A IN Table | *val* op (SELECT COUNT(*) FROM Table)

op ::= = | < | >

Figure 2: Queries supported by PrivateSQL. The terminal R corresponds to one of the base relations in the schema, the terminal A corresponds to an attribute in the schema and *val* is a value in the domain of an attribute.

4. PRIVACY MODEL

Privacy for a Single Relation: The formal definition of differential privacy (DP) considers single relation databases:

Definition 4.1 (DP for Single Relation) *A mechanism $\mathcal{M} : dom(R) \rightarrow \Omega$ is ϵ-differentially private if for any relational database instance $D \in dom(R)$ of size at least 1 and $D' = D - \{t\}$, and $\forall O \subseteq \Omega$:*

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

This definition implies that deleting a row from a database does not significantly change the probability that the output of the mechanism lies in a specific set. This is equivalent to the standard definition of differential privacy [15].

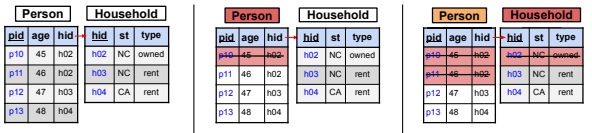
However, defining privacy for a schema with multiple relations is more subtle. First, we need to determine which relation(s) in the schema is(are) private. Second, adding or removing a record in a relation can cause the addition and/or removal of multiple rows in other relations due to schema constraints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

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

To capture privacy policies and key constraints, we propose a definition of neighboring tables inspired by Blowfish privacy [23]. For two database instances $\mathbf{D}$ and $\mathbf{D}'$, we say that $\mathbf{D}$ is a *strict superset* of $\mathbf{D}'$ (denoted by $\mathbf{D} \sqsupset \mathbf{D}'$) if (a) $\forall i, D_i \supseteq D'_i$ and (b) $\exists i, D_i \supset D'_i$. That is, all records that appear in $\mathbf{D}'$ also appear in $\mathbf{D}$ and there is at least one row in a relation of $\mathbf{D}$ that does not appear in $\mathbf{D}'$.

Definition 4.2 (Neighboring Databases) *Given a schema $\mathbb{S}$ with a set of foreign key constraints $\mathcal{C}$, and a privacy policy $P = (R_i, \epsilon)$, for a valid database instance $\mathbf{D} = (D_1, \ldots, D_k) \in dom(\mathbb{S}, \mathcal{C})$, we denote by $\ominus_{\mathcal{C}}(\mathbf{D}, R_i)$ a set of databases such that $\forall \mathbf{D}' \in \ominus_{\mathcal{C}}(\mathbf{D}, R_i)$:*

- *$\exists r \in D_i$, but $r \notin D'_i$, and*
- *$\mathbf{D}'$ satisfies $\mathcal{C}$, and*



(a) A database instance of the Census schema.

(b) Neighboring DB instance under Person policy.

(c) Neighboring DB instance under Household policy.

Figure 3: Neighboring databases under foreign key constraints.

- $\nexists \mathbf{D}''$ *that satisfies* $\mathcal{C}$ *and* $\mathbf{D} \supset \mathbf{D}'' \supset \mathbf{D}'$.

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

Example 1. *Consider the database of Fig. 3a with schema* Person *(pid, age, hid) and* Household *(hid, st, type).* Person.*hid is a foreign key to* Household*. Fig. 3b shows a neighboring instance of the original database under privacy policy* $P = (\text{Person}, \epsilon)$*. Notice that in that instance, the* Household *table is unchanged and only person* $p10$ *is removed. However, under the privacy policy* $P = (\text{Household}, \epsilon)$ *(Fig. 3c) removing* $h02$ *from* Household *results in deleting two rows in* Person *table. In this case, neighboring databases differ in both the primary private relation* Household *as well as a secondary private relation* Person*.*

Definition 4.3 (Secondary Private Relations) *Let* $\mathbb{S}$ *be a schema with constraints* $\mathcal{C}$ *and* $P = (R_i, \epsilon)$ *be a privacy policy. Then a relation* $R_j \in \mathbb{S}$ *is a secondary private relation iff:* $\exists \mathbf{D} \in dom(\mathbb{S}, \mathcal{C}), \exists \mathbf{D}' \in \ominus_{\mathcal{C}}(\mathbf{D}, R_i)$ *s.t.* $\mathbf{D}_j \neq \mathbf{D}'_j$.

We call a policy that results in no secondary private relations (e.g., Fig. 3b) a *simple* policy. In this case, neighboring tables differ in only the primary private relation in exactly one row. We call policies that result in secondary private relations (e.g., Fig. 3c) as *complex* policies.

Definition 4.4 (DP for Multiple Relations) *Given a schema* $\mathbb{S}$ *with foreign key constraints* $\mathcal{C}$ *and privacy policy* $P = (\mathtt{R}, \epsilon)$ *be a policy. A mechanism* $\mathcal{M} : dom(\mathbb{S}, \mathcal{C}) \to \Omega$ *is* P*-differentially private if for every set of outputs* $O \subseteq \Omega$*,* $\forall \mathbf{D} \in dom(\mathbb{S}, \mathcal{C})$*, and* $\forall \mathbf{D}' \in \ominus_{\mathcal{C}}(\mathbf{D}, \mathtt{R})$*:*

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

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

Theorem 4.1 (Sequential Composition) *Given a schema* $\mathbb{S}$ *with constraints* $\mathcal{C}$*, let mechanisms* M_1*,* M_2 *that satisfy* P_1*-DP and* P_2*-DP, with* $P_i = (\mathtt{R}, \epsilon_i)$*. Then the sequence of* M_1 *and* M_2 *satisfies* P_{seq}*-DP, with* $P_{seq} = (\mathtt{R}, \epsilon_1 + \epsilon_2)$.

Global Sensitivity: Designing differentially private mechanisms requires an important notion called *global sensitivity* – the maximum change to the query output in neighboring datasets. In multi-relational databases, the sensitivity of a query can change depending on which relation is identified as the primary private relation. We denote by Δ_R the sensitivity of a query with respect to relation $R \in \mathbb{S}$.

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

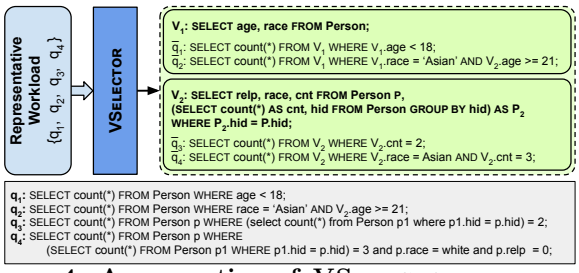


Figure 4: An execution of VSelector on a workload of 4 queries, producing two distinct views.

Definition 4.5 (Global Sensitivity for View) *Given a schema* $\mathbb{S}$ *with foreign key constraints* $\mathcal{C}$ *and privacy policy* $P = (\mathtt{R}, \epsilon)$*. A view query* V *takes as input an instance* $\mathbf{D}$ *and outputs a relation instance* $V(\mathbf{D})$*. The global sensitivity of* V *w.r.t.* $\mathtt{R}$ *is defined as the maximum number of rows that change in* V *across neighboring databases w.r.t.* $\mathtt{R}$*, i.e.,*

$$\Delta^{\mathcal{C}}_{\mathtt{R}}(V) = \max_{\mathbf{D} \in dom(\mathbb{S}, Q)} \Delta^{\mathcal{C}}_{\mathtt{R}}(V, \mathbf{D}) \quad (1)$$

$$\text{where, } \Delta^{\mathcal{C}}_{\mathtt{R}}(V, \mathbf{D}) = \max_{\mathbf{D}' \in \ominus_{\mathcal{C}}(\mathbf{D}, \mathtt{R})} V(\mathbf{D}) \triangle V(\mathbf{D}') \quad (2)$$

is the down sensitivity of a given instance $\mathbf{D}$ *and* $A \triangle B = (A \setminus B) \cup (B \setminus A)$ *denotes symmetric difference.*

Relationship to Other Privacy Notions: Most variants of differential privacy that apply to relational data can be captured using a single private relation and foreign key constraints on an acyclic schema [5, 9, 14, 25, 26, 29]. For instance, a graph $G = (V, E)$ can be represented as a schema with relations Node(id) and Edge(src_id, dest_id) with foreign key references from Edge to Node (src_id $\to$ id and dest_id $\to$ id). Edge-DP [25] is captured by P-DP by setting Edge as the primary private relation $\mathtt{R}$, Node-DP [26] is captured if we set Node as $\mathtt{R}$. Under the latter policy, neighboring databases differ in one row from Node and all rows in Edge that refer to the deleted Node rows. Similarly, user-level- and event-level-DP are also captured using a database schema User(id, ...), Event(eid, uid, ...) with events referring to users via a foreign key (uid $\to$ id). By setting the Event (User) as the primary private relation, we get Event-DP (User-DP, resp.) [14].

The privacy model in FLEX [24] considers neighboring tables that differ in *exactly* one row in one relation. FLEX does not capture standard variants of DP described above since the FLEX privacy model *ignores all constraints* in the schema. For instance, using FLEX for graphs would consider neighboring databases that differ in exactly one edge or one node, but never in all the edges connected to a node. Thus, FLEX's privacy model can not capture Node-DP.

5. VIEW SELECTION

The view selection module VSelector takes as input a set of *representative queries* $\mathcal{Q}$ over a schema $\mathbb{S}$ and outputs a set of views $\mathcal{V}$ such that every query $q \in \mathcal{Q}$ is *linearly answerable* using some $V \in \mathcal{V}$.

Definition 5.1. *A query* q *over schema* $\mathbb{S}$ *is answerable using a view* V *if there is a query* $\bar{q}$ *defined on the attributes in* V *such that for all database instances* $\mathbf{D} \in dom(\mathbb{S})$*, we have,* $q(\mathbf{D}) = \bar{q}(V(\mathbf{D}))$*. Additionally, we say that* q *is linearly answerable using* V*, if* $\bar{q}$ *is linear on* V.

Linear answerability ensures that queries in $\mathcal{Q}$ can be directly answered from some $V \in \mathcal{V}$ without additional join or

group-by operations. Moreover, the privacy analysis of sets of linear queries is easy and allows the use of well known workload aware algorithms in the PrivSynGen module.

Fig. 4 shows an execution of VSelector, which for input $\mathcal{Q}$ produces view V_1 and V_2. Queries q_1 and q_2 can be answered by the linear queries $\bar{q}_1$ and $\bar{q}_2$ on V_1. Similarly, q_3 and q_4 can be answered from queries $\bar{q}_3$ and $\bar{q}_4$ on V_2.

Approach: We propose a heuristic algorithm VSelector with the following properties: (a) every $q \in \mathcal{Q}$ is linearly answerable using one $V \in \mathcal{V}$, (b) all queries mapped to a view have the same sensitivity, and (c) as many queries as possible are mapped to the same view. These properties help minimize the noise added to ensure differential privacy.

VSelector first applies a query transformation function, denoted by MapQueryToView, that takes as input a query q and returns a query-view pair $(\bar{q}, V)$. Then, all pairs with a common view are grouped together, such that each view V is associated with a set of transformed queries Q_V. This is followed by a step of *attribute pruning* where a view V retains only attributes appearing in a query of Q_V. In Fig. 4 we see a full execution of VSelector on a workload of 4 queries, resulting in views V_1 and V_2 with partial workloads $Q_{V_1} = \{\bar{q}_1, \bar{q}_2\}$ and $Q_{V_2} = \{\bar{q}_3, \bar{q}_4\}$ respectively.

The MapQueryToView function consists of 3 sequential steps: (a) the baseline transformation, (b) decorrelation, and (c) moving filter operations. Following the grammar of Fig. 2 the baseline transformer takes as input a SQL query q ::= AggQuery and returns V ::= select AttrList from TableList and $\bar{q}$ ::= select count(*) from v. Next, the query transformer performs decorrelation [6] by rewriting correlated subqueries of views in terms of joins. Finally, we move filtering operations from the view V to the query $\bar{q}$. To illustrate how the MapQueryToView works consider query q_3 from the example of Fig. 4 that contains a correlated subquery, that is transformed to the pair $(V_2, \bar{q}_3)$.

6. VIEW SENSITIVITY ANALYSIS

Computing the global sensitivity of a SQL view (lines 6-7 of Algorithm 1) is a hard problem [5], as single changes in a base relation could affect a large (or even unbounded) number records in the view. Moreover, complex privacy policies resulting in secondary private relations (see Definition 4.3), further complicate sensitivity estimation.

In this section we focus on *simple privacy policies* resulting only in a primary private relation in the schema and discuss complex policies in Section 7. Section 6.1 describes SensCalc a rule-based algorithm that computes the *constraint-oblivious* down sensitivity of a a view V on a database instance $\mathbf{D}$. Section 6.2 describes how to rewrite a view using truncation operators so that for simple privacy policies, the sensitivity output by SensCalc is indeed the global sensitivity of the rewritten view V^τ (see Theorem 6.1). Section 6.3 presents a DP method for learning thresholds needed for truncation operators.

We assume w.l.o.g. that a view V is expressed in relational algebra. The algebra expression can be viewed as a tree, where internal nodes are algebra operators and the leaf nodes are base relations of $\mathbb{S}$.

6.1 Sensitivity Calculator

SensCalc, computes the following variant of down sensitivity that captures the maximum change caused by *removing* any one tuple from the primary private relation R.

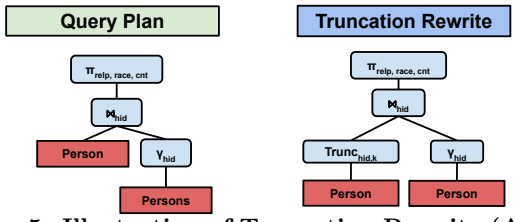


Figure 5: Illustration of Truncation Rewrite (Alg 2)

Definition 6.1 (Constraint-Oblivious Down Sensitivity) *Given schema $\mathbb{S}$ and a privacy policy $(\mathtt{R}, \epsilon)$, the constraint-oblivious down sensitivity of V given $\mathbf{D}$ w.r.t. R, denoted by $\Delta_{\mathtt{R}}(V, \mathbf{D})$, is defined as the maximum number of rows that change in V when removing a row from R.*

$$\Delta_{\mathtt{R}}(V, \mathbf{D}) = \max_{r \in dom(\mathtt{R})} V(\mathbf{D}) \triangle V(\mathbf{D} - \{r\}), \qquad (3)$$

where $\mathbf{D} - \{r\}$ means removing tuple r from instance $\mathbf{D}$.

For simple privacy policies, the constraint-oblivious down sensitivity is equivalent to the down sensitivity (defined in Section 4 Eq. (4)), i.e., for any simple policy P and any V: $\Delta_{\mathtt{R}}(V, \mathbf{D}) = \Delta^{C}_{\mathtt{R}}(V, \mathbf{D})$. Combined with truncation rewrites described later, the sensitivity output by SensCalc will be the right global sensitivity for simple policies.

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

Given an input view V and $\widehat{\mathtt{mf}}$, the sensitivity calculator computes $\hat{\Delta}_{\mathtt{R}}(V, \widehat{\mathtt{mf}})$ by a recursive application of the rules in Table 1 to each subexpression S of V. The bounds at the base relations are as follows: the sensitivity bounds $\hat{\Delta}_{\mathtt{R}}(\mathtt{R}) = 1$ and $\hat{\Delta}_{\mathtt{R}}(R) = 0$ for $R \in \mathbb{S} - \{\mathtt{R}\}$ and the max-frequency bounds are supplied by $\widehat{\mathtt{mf}}$. In Table 1 we summarize the rules of SensCalc. Operators such as project, select, and groupby do not increase the sensitivity bound of their input relation, while groupby-count doubles it. equijoin results in relations with higher sensitivity bounds compared to its inputs. In terms of the mf bounds, most unary operators shown in Table 1 have unchanged mf. Note that we restrict the equijoin operator to join on attributes from the base relations in $\mathbb{S}$. The last row refers to a *truncation operator*, which is described in Section 6.2.

These rules are similar to those of *elastic sensitivity* [24], but with some key differences that allow for a tighter sensitivity analysis. SensCalc uses additional rules using *keys*, as shown in the last column of Table 1. The new rules keep track of key constraints through operators. This allows the addition of new rules for joins on key attributes that permit lower sensitivity bounds than a standard join, as illustrated in the following example.

Example 2 (Sensitivity Calculation) *Consider calculating the sensitivity of V_2 from Fig. 4 under Person policy. A relational algebra expression for view V_2 is (Fig. 5 (left))*

$$\pi_{race,relp,cnt}(\mathit{Person} \bowtie_{hid} (\gamma^{COUNT}_{hid}(\mathit{Person}))).$$

V_2 has a row for each person reporting the person's race, relp, and size of their household. SensCalc *initializes* $\hat{\Delta}_{\mathtt{R}}(\mathit{Person})$

Table 1: Update rules for sensitivity and max-frequency bounds. New rules are shaded.

Operators	Sensitivity Bound $\hat{\Delta}_{\mathtt{R}}(S)$		Maximum Frequency Bound $\widehat{\mathtt{mf}}(\mathbf{A}', S), \mathbf{A}' \subseteq attr(S)$	Key Set $Keys(S)$
$S = \pi_{\mathbf{A}}(R)$	$\hat{\Delta}_{\mathtt{R}}(R)$		$\widehat{\mathtt{mf}}(\mathbf{A}', R)$	$\{\mathbf{A}' \subseteq attr(S) \mid \mathbf{A}' \in Keys(R)\}$
$S = \sigma_{\phi}(R)$	$\hat{\Delta}_{\mathtt{R}}(R)$		$\widehat{\mathtt{mf}}(\mathbf{A}', R)$	$\{\mathbf{A}' \subseteq attr(S) \mid \mathbf{A}' \in Keys(R)\}$
$S = \gamma_{\mathbf{A}}(R)$	$\hat{\Delta}_{\mathtt{R}}(R)$		$\widehat{\mathtt{mf}}(\mathbf{A}', R)$	$\{\mathbf{A}\} \cup \{\mathbf{A}' \subseteq attr(S) \mid \mathbf{A}' \in Keys(R)\}$
$S = \gamma_{\mathbf{A}}^{COUNT}(R)$	$2\hat{\Delta}_{\mathtt{R}}(R)$		$\widehat{\mathtt{mf}}(\mathbf{A}', R)$	$\{\mathbf{A}\} \cup \{\mathbf{A}' \subseteq attr(S) \mid \mathbf{A}' \in Keys(R)\}$
$S = R_1 \bowtie_{\mathbf{A}_1=\mathbf{A}_2} R_2$ or $S = R_1 \ltimes_{\mathbf{A}_1=\mathbf{A}_2} R_2$ where $\mathbf{A}_1, \mathbf{A}_2$ are from $\mathbb{S}$	General case	$\widehat{\mathtt{mf}}(\mathbf{A}_1, R_1) \cdot \hat{\Delta}_{\mathtt{R}}(R_2) + \widehat{\mathtt{mf}}(\mathbf{A}_2, R_2) \cdot \hat{\Delta}_{\mathtt{R}}(R_1) + \hat{\Delta}_{\mathtt{R}}(R_1) \cdot \hat{\Delta}_{\mathtt{R}}(R_2)$	$\max(\widehat{\mathtt{mf}}(\bar{\mathbf{A}}_2, R_1) \cdot \widehat{\mathtt{mf}}(\mathbf{A}_2, R_1), \widehat{\mathtt{mf}}(\bar{\mathbf{A}}_1, R_2) \cdot \widehat{\mathtt{mf}}(\mathbf{A}_1, R_2))$ where $\bar{\mathbf{A}}_i = \mathbf{A}' - attr(R_i)$	$\{\mathbf{A}' \in Keys(R_2) \mid \mathbf{A}_1 \in Keys(R_1)\} \cup \{\mathbf{A}' \in Keys(R_1) \mid \mathbf{A}_2 \in Keys(R_2)\}$
	No common ancestors	$\max(\widehat{\mathtt{mf}}(\mathbf{A}_1, R_1) \cdot \hat{\Delta}_{\mathtt{R}}(R_2), \widehat{\mathtt{mf}}(\mathbf{A}_2, R_2) \cdot \hat{\Delta}_{\mathtt{R}}(R_1))$		
	Join on key ($\mathbf{A}_1 \in Keys(R_1)$)	$\widehat{\mathtt{mf}}(\mathbf{A}_2, R_2) \cdot \hat{\Delta}_{\mathtt{R}}(R_1) + \hat{\Delta}_{\mathtt{R}}(R_2)$		
$S = \tau_{\mathbf{A},k}(R)$	$k \cdot \hat{\Delta}_{\mathtt{R}}(R)$		$\min\{k, \widehat{\mathtt{mf}}(\mathbf{A}', R)\}$ if $\mathbf{A} \subseteq \mathbf{A}'$; $\widehat{\mathtt{mf}}(\mathbf{A}', R)$, o.w.	$\{\mathbf{A}' \subseteq S \mid \mathbf{A}' \in Keys(R)\}$

Algorithm 2 Truncation Rewrite $(V, \mathtt{R}, \mathbf{k})$

```
Initialize V^τ ← V
for every path p_l from leaf relation R_l to root in V do
    for every R_1 ⋈_{A_1=A_2} R_2 on p_l, where A_1 ⊆ attr(R_l) do
        ▷(semijoin is also treated as a special equijoin)
        if A_1 ∉ Keys(R_1) and R is a base relation of R_2 then
            k ← k_{A_1}
            Insert τ_{A_1,k}(R_l) above R_l in V^τ
            A ← A ∪ (A_1)
Return V^τ
```

to 1 and applies the rules of Table 1 bottom up. First the GROUPBY-COUNT *operator is processed, resulting in* $S = \gamma_{hid}^{COUNT}(\mathsf{Person})$ *with* $\hat{\Delta}_{\mathtt{R}}(S) = 2 \cdot \hat{\Delta}_{\mathtt{R}}(\mathsf{Person}) = 2$ *and* S *has hid as a key. Next, the* EQUIJOIN *operator is processed, joining on key hid of* S*, producing* $S_{\bowtie} = \mathsf{Person} \bowtie_{hid} S$ *with:* $\hat{\Delta}_{\mathtt{R}}(S_{\bowtie}) = F \cdot \hat{\Delta}_{\mathtt{R}}(S) + \hat{\Delta}_{\mathtt{R}}(\mathsf{Person}) = F \cdot 2 + 1$ *where* $F = \widehat{mf}(hid, \mathsf{Person})$*. Note that without the "Join on key" rule, the bound would be* $(F \cdot 3 + 2)$*. This difference is only exacerbated for views with more joins. Last, the* PROJECTION *operator is processed, leaving the bound unchanged.*

Given $\mathbf{D}, V$ and upper bounds on max-frequency $\widehat{\mathtt{mf}}$, we can show that $\hat{\Delta}_{\mathtt{R}}(V, \widehat{\mathtt{mf}})$ calculated by SENSCALC is an upper bound on $\Delta_{\mathtt{R}}(V, \mathbf{D})$, and thus an upper bound on the down sensitivity $\Delta_{\mathtt{R}}^{\mathcal{C}}(V, \mathbf{D})$ for simple policies.

6.2 Bounding Sensitivity via Truncations

As shown in Example 2, the sensitivity bounds produced by the SENSCALC can be dependent on the max-frequency bounds on base relations. We now show how to add *truncation operators* to the view expression. These operators delete tuples that contain an attribute combination appearing in a join and whose frequency exceeds a *truncation threshold* k specified in the operator.

Definition 6.2 (TRUNCATION OPERATOR) *The truncation operator* $\tau_{\mathbf{A},k}(R)$ *takes in a relation* R*, a set of attributes* $\mathbf{A} \subseteq attr(R)$ *and a threshold* k *and for all* $a \in dom(\mathbf{A})$*, if* $f(a, \mathbf{A}, R) > k$*, then any* r *from* R *with* $r[\mathbf{A}] = a$ *is removed.*

Truncation rewrite (see Algorithm 2) adds truncation operators to V and forms a new query plan V^{τ}. The algorithm takes as input a view V, a primary private relation $\mathtt{R}$, and a vector of truncation thresholds $\mathbf{k}$, indexed by the attribute subset to which the threshold applies. It traverses every path p_l from relation R_l to the root operator and every join $R_1 \bowtie_{\mathbf{A}_1=\mathbf{A}_2} R_2$ on this path. If one of the join attributes is from R_l—say $\mathbf{A}_1 \subseteq R_l$—and $\mathbf{A}_1$ is not a key for R_1 and the primary private table $\mathtt{R}$ appears as a base relation in the expression R_2, then we insert $\tau_{\mathbf{A}_1,k}(R_l)$ above R_l in V^{τ}.

Algorithm 3 LEARNTHRESHOLD $(\mathbf{D}, V^{\tau}, \theta, \epsilon_{mf})$

```
Traverse operators in V^τ from leaf to root and add each truncation operator to T if it is not in the list.
for τ_{A,k}(R) ∈ T do
    q'_i ← sub-tree at τ_{A,k}(R) ∈ V^τ        ▷ Truncate at k = i
    Q ← {(|q'_i| − |R|·θ)/i | i = 1, 2, ...}
    Set i ← SVT(D, Q, 0, ε_mf/|T|) as the truncation threshold for τ_{A,k}(R)
```

EXAMPLE 3. *Fig. 5 (right) shows the truncation operators are inserted before* Person *relation. The truncation operators cut down the maximum frequency of hid to* k *so that the sensitivity bound can be bounded by* $3k$*, even when* $\widehat{mf}$ *for household id in* Person *is unbounded. In this case,* $\hat{\Delta}_{\mathtt{R}}(S_{\bowtie}) = k \cdot \hat{\Delta}_{\mathtt{R}}(\gamma_{hid}^{COUNT}(\mathsf{Person})) + \hat{\Delta}_{\mathtt{R}}(\tau_{hid,k}(\mathsf{Person})) = k \cdot 2 + k = 3k$*.*

After truncation rewrite is applied, the estimated sensitivity no longer depends on $\widehat{\mathtt{mf}}$, but rather on the truncation thresholds. If the thresholds are set in a data independent manner, or using a DP algorithm (as discussed in Section 6.3) we can show that the sensitivity output by SENSCALC on V^{τ} is the global sensitivity for simple policies.

THEOREM 6.1. *Consider a schema* $\mathbb{S} = (R_1, \ldots, R_k)$ *with foreign constraints* $\mathcal{C}$*, and simple privacy policy* $(\mathtt{R}, \epsilon)$*. For any* V*, let* V^{τ} *denote the truncation rewrite of* V *using a fixed set of truncation thresholds* $\mathbf{k}$ *(Algorithm 2). The global sensitivity of* V^{τ} *is bounded by* SENSCALC*:*

$$\Delta_{\mathtt{R}}^{\mathcal{C}}(V^{\tau}) = \Delta_{\mathtt{R}}(V^{\tau}) \leq \hat{\Delta}_{\mathtt{R}}(V^{\tau}).$$

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

The truncation rewrite introduces bias: i.e., $\exists \mathbf{D}, V(\mathbf{D}) \neq V^{\tau}(\mathbf{D})$. However, the global sensitivity computed after truncation is usually much smaller reducing error due to noise. We empirically measure the effect of truncation bias in Section 9.4. Our truncation methods are related to *Lipschitz extension* techniques which also tradeoff bias for noise typically by truncating the data. Existing methods apply to specific queries on graphs [10,12,21,25,26] or only on monotone queries [9]. Our technique applies to general relational data and more complex queries.

6.3 Learning Truncation Thresholds

We propose LEARNTHRESHOLD (Algorithm 3), an algorithm for privately learning truncation thresholds. It takes as input a query plan, V^{τ}, a data instance $\mathbf{D}$, the privacy parameter ϵ_{mf}, and a parameter θ controlling the number

of tuples preserved. LearnThreshold works in a bottom-up manner to identify the ordered list T of unique truncation operators in V^τ. For each truncation operator $\tau_{\mathbf{A},k}(R)$, let q'_i be the sub-query rooted at the operator if truncation threshold k is set to be i. We consider a stream of queries $Q = \{q_i \mid i = 1, 2, \ldots\}$, where $q_i = (|q'_i(\mathbf{D})| - |R| \cdot \theta)/i$ measures whether θ fraction of R can be preserved if truncating R at threshold i. The sensitivity of q_i is bounded by the sensitivity of R, which is bounded since the LearnThreshold operates bottom-up. We apply the *sparse vector technique* [15] which returns the first i such that $q_i(\mathbf{D}) > 0$ with the given privacy budget $\epsilon_{mf}/|T|$.

7. HANDLING COMPLEX POLICIES

We now shift our focus on computing view sensitivity for *complex privacy policies*. Recall that under complex privacy policies, neighboring databases differ in the primary private relation as well as other secondary private relations (see Fig. 3c). Due to this, the constraint oblivious down sensitivity is not the same as the down sensitivity (i.e., $\Delta_{\mathtt{R}}(V, \mathbf{D}) \neq \Delta^{\mathcal{C}}_{\mathtt{R}}(V, \mathbf{D})$). Moreover, removing a row in the primary relation might result in an unbounded number of rows deleted in the secondary private relation – e.g., under Household policy the maximum change in Person is unbounded in the absence of external information. Truncation operators discussed previously only limit the frequencies of attributes involved in joins, but not the change in secondary private relations.

We now present the *semijoin rewrite* that transforms view V into $V^\ominus$ so that the sensitivity computed by SensCalc on $V^\ominus$ equals its down sensitivity: $\Delta_{\mathtt{R}}(V^\ominus, \mathbf{D}) = \Delta^{\mathcal{C}}_{\mathtt{R}}(V^\ominus, \mathbf{D})$.

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

Definition 7.1 (Transitive Referral) *A relation S transitively refers to a relation R through foreign keys if there exists a relation T such that $S.A \rightarrow T.B$ and T transitively refers to relation R through foreign keys. Moreover, a row $s \in S$ transitively refers to a row $r \in R$ if there is a row $t \in T$ such that $s \rightarrow t$ and t transitively refers to r. If s transitively refers to r, we denote that $s \twoheadrightarrow r$.*

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

Theorem 7.1 (Transitive Deletion) *Given an acyclic schema $\mathbb{S} = (R_1, \ldots, R_k)$ with foreign key constraints $\mathcal{C}$, and a privacy policy (R_i, ϵ). For $\mathbf{D} \in dom(\mathbb{S}, \mathcal{C})$ and $r \in D_i$, we denote $\ominus_{\mathcal{C}}(\mathbf{D}, (r, R_i)) = (D_1^\ominus, D_2^\ominus, \ldots, D_k^\ominus)$, where $D_j^\ominus = D_j - \{t | t \in D_j, t \twoheadrightarrow r\}$. Then we have:*

$$\ominus_{\mathcal{C}}(\mathbf{D}, R_i) = \cup_{r \in D_i} \ominus_{\mathcal{C}} (\mathbf{D}, (r, R_i)).$$

Based on this theorem, the down sensitivity of a view (defined in Definition 4.5) can be expressed as:

$$\Delta^{\mathcal{C}}_{\mathtt{R}}(V, \mathbf{D}) = \max_{r \in dom(\mathtt{R})} V(\mathbf{D}) \triangle V(\ominus_{\mathcal{C}}(\mathbf{D}, (r, \mathtt{R}))). \quad (4)$$

Semijoin Rewrite: Our proposed rewrite works in two steps. First, it replaces every secondary private base relation R_j in V with a semijoin expression (Eq. (5)) that makes explicit the transitive dependence between the primary private relation R and R_j. The resulting expression $V^\ltimes$ is such that $V(\mathbf{D}) = V^\ltimes(\mathbf{D})$. Moreover, the down sensitivity is now correct $\Delta_{\mathtt{R}}(V^\ltimes, \mathbf{D}) = \Delta^{\mathcal{C}}_{\mathtt{R}}(V^\ltimes, \mathbf{D})$ since transitive deletion is captured by the semijoin expressions.

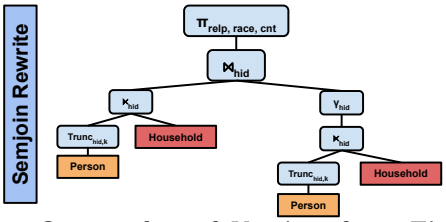


Figure 6: Query plan of V_2 view from Fig. 5, after adding semijoin operators, where Household is the primary relation.

Second, to handle the high sensitivity of secondary private base relations, we add truncation operations using (Algorithm 2) to the semijoin expressions and transform $V^\ltimes$ to $V^\ominus$. More formally,

Definition 7.2 (Semijoin Rewrite) *The semijoin rewrite 1) takes as input V and transforms it into $V^\ltimes$ such that $V^\ltimes$ is identical to V except that each base relation R_j of V is replaced with $R_j^\ltimes$, which is recursively defined as:*

$$R_j^\ltimes = \begin{cases} R_j, & \text{if } R_j = \mathtt{R} \\ (((R_j \ltimes R^\ltimes_{p(j)_1}) \ltimes R^\ltimes_{p(j)_2}) \ldots \ltimes R^\ltimes_{p(j)_\ell}) & \text{else} \end{cases} \quad (5)$$

where each relation $S \in \{R_{p(j)_1}, R_{p(j)_2}, \ldots, R_{p(j)_\ell}\}$ is such that: (a) R_j refers to S, and (b) $S = \mathtt{R}$ or transitively refers to the primary private relation R through foreign keys.

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

Lemma 7.1. *Given an acyclic schema $\mathbb{S}$ with foreign key constraints $\mathcal{C}$, privacy policy $P = (\mathtt{R}, \epsilon)$, and a view V. Let $V^\ltimes$, $V^\ominus$ be as defined in Definition 7.2. Then, for any database instance $\mathbf{D} \in dom(\mathbb{S}, \mathcal{C})$, we have $V(\mathbf{D}) = V^\ltimes(\mathbf{D})$ and the down sensitivity of $V^\ominus$ equals the constraint-oblivious down sensitivity of $V^\ominus$:*

$$\Delta^{\mathcal{C}}_{\mathtt{R}}(V^\ominus, \mathbf{D}) = \Delta_{\mathtt{R}}(V^\ominus, \mathbf{D}) \quad (6)$$

Putting it all together: Given a view V, we first apply Algorithm 2 to V to add truncation operators to the primary private relation R and obtain V^τ. Then we run semijoin rewrite in Definition 7.2 to get $V^{\tau,\ominus}$. As the second step of semijoin rewrite introduces extra truncation operators into the query plan, existing truncation operators may become redundant, in which case we keep ones closest to the base relation. The following example shows the entire procedure of a view rewrite.

Example 4. *Recall the query plan V and its truncation rewrite V^τ from Fig. 5. Under the Household policy, Person is a secondary private relation. As shown in Fig. 6 the semijoin rewrite will replace the Person relations in V^τ with a semijoin between Person and Household. Truncation operators are also added to bound the sensitivity of the Person table to get $V^{\tau,\ominus}$. Note that the truncation operator in V^τ is redundant in $V^{\tau,\ominus}$ and removed since the semijoin rewrite introduces the same truncation operator on Person.*

Theorem 7.2 shows that after applying the truncation and semijoin rewrites the sensitivity of $V^{\tau,\ominus}$ output by SensCalc is the global sensitivity. Proof follows from Theorem 6.1 and Lemma 7.1.

THEOREM 7.2. *Given an acyclic schema $\mathbb{S} = (R_1, \ldots, R_k)$ with foreign constraints $\mathcal{C}$, and $\mathtt{R} \in \mathbb{S}$. For any V, let $V^{\tau,\ominus}$ denote V after applying both the truncation rewrite (Algorithm 2) and the semijoin rewrite (Definition 7.2) , where the truncation thresholds are* $\mathbf{k}$ *and are fixed. The global sensitivity of $V^{\tau,\ominus}$ is bounded:*

$$\Delta^{\mathcal{C}}_{\mathtt{R}}(V^{\tau,\ominus}) \leq \hat{\Delta}_{\mathtt{R}}(V^{\tau,\ominus}).$$

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

8. GENERATING SYNOPSES

In this section we describe how PrivateSQL generates private synopses. More specifically, we describe how give a view definition PrivateSQL generates differentially private synopses and how privacy budget is split across views. We end this section with an end-to-end privacy analysis.

Private Synopsis Generator The PrivSynGen module produces a private synopsis of a single materialized view. The input to PrivSynGen is a materialized view $V(\mathbf{D})$, a set of linear (on V) queries Q_V, and a privacy budget ϵ_V. Its output is $\tilde{D}_V$, an ϵ_V-DP synopsis of $V(\mathbf{D})$.

This component is probably the most well understood as it is an instance of a common problem studied in the DP literature – answering a set of linear queries on a single table [22, 31, 39]. Furthermore, synopsis generators can be *workload aware* or *workload agnostic* depending on whether they optimize their output w.r.t. a set of linear queries Q_V.

We use both workload-agnostic and workload-aware instances of PrivSynGen, returning a vector of counts. More specifically, we use: W-nnls, a workload-aware version of non-negative least squares inference [28], and the workload-agnostic algorithms Identity and Part, the latter of which performs the partitioning step of the DAWA algorithm [27].

Privacy Budget Allocator Recall from Definition 4.5 that changing a row in the primary sensitive relation R results in changing $\Delta_{\mathtt{R}}(V)$ rows in view V, where $\Delta_{\mathtt{R}}(V)$ is the sensitivity of view V. Thus, running an ϵ_V-DP algorithm on view V will satisfy $(\mathtt{R}, \Delta_{\mathtt{R}}(V) \cdot \epsilon_V)$-DP. For that reason the any budget allocation strategy for materializing views needs to take into account the sensitivity of each view. In PrivateSQL, budget allocation is performed by BudgetAlloc, which has access to the intermediate non-private outputs of PrivateSQL and returns $\mathcal{E} = \{\epsilon_V\}_{V \in \mathcal{V}}$, a budget allocation that satisfies:

$$\sum\nolimits_{V \in \mathcal{V}} \hat{\Delta}_V \epsilon_V \leq \epsilon, \tag{7}$$

where $\hat{\Delta}_V$ is an upper bound of $\Delta_{\mathtt{R}}(V)$ as computed from SensCalc (see Section 6.1). The ideal allocator would be a *query fair* allocator that splits the budget such that each query of the representative workload incurs the same error. In this work, we consider allocators of the following form:

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

As long as $\forall V \in \mathcal{V} : \lambda_V \geq 0$ and $\sum_{V \in \mathcal{V}} \lambda_V \leq 1$ this satisfies Eq. (7). We use 4 strategies for allocating budget – Naive divides ϵ equally among views; Wsize, splits the privacy budget according to the size of Q_V the partial workload of each view; Wsens allocates the privacy budget according to the sensitivity of each Q_V; and Vsens splits the privacy budget proportionally to the sensitivity of each view.

Table 2: PrivateSQL and input options used.

Census Input	Options
Dataset	**Census$_{NC}$**, Census$_{PM}$
Privacy Policy	**Person**, Household
Privacy Budget ϵ	2.0, **1.0**, 0.5, 0.25, 0.125
Representative Workload	$\boldsymbol{W_1}$, W_2, W_1', W_2'
Query Workload	$\boldsymbol{W_1}$, W_2
TPC-H Input	**Options**
Dataset	**TPC-H**
Privacy Policy	**Customer**
Privacy Budget ϵ	2.0, **1.0**, 0.5, 0.25, 0.125
Representative Workload	$\boldsymbol{W_3}$
Query Workload	$\boldsymbol{W_3}$
PrivateSQL Config.	**Options**
BudgetAlloc	**WSize**, WSens, Naive, VSens
PrivSynGen	**W-nnls**, Identity, Part

Privacy We conclude with a formal privacy statement.

THEOREM 8.1. *Given an acyclic schema $\mathbb{S} = (R_1, \ldots, R_k)$ with foreign constraints Q and a privacy policy $P = (\epsilon, \mathtt{R})$, where $\mathtt{R} \in \mathbb{S}$.* PrivateSQL *satisfies P-differential privacy.*

9. EXPERIMENTS

We evaluate PrivateSQL on both a use case inspired by U.S. Census data releases as well as the TPC Benchmark H(TPC-H) [3]. In Section 9.2 we present an end-to-end error evaluation analysis. In Section 9.3, we compare with prior work (Flex [24]). Lastly, in Section 9.4 we evaluate alternative choices for components of PrivateSQL.

9.1 Setup

Table 2 summarizes settings with defaults in boldface.

Datasets: We use the public synthetic U.S. Census dataset [36] with the following schema: person(id, sex, gender, age, race, hid) and household(hid, location). We create two datasets from the full Census data by filtering on location: Census$_{PM}$ limits to a specific PUMA region (a region roughly the size of a town) and Census$_{NC}$ limits to locations within North Carolina. Census$_{PM}$ contains $50K$ and $38K$ tuples in Person and Household respectively, while Census$_{NC}$ contains $5.4M$ and $2.7M$ tuples, resp. We also use the TPC-H benchmark with a schema consisting of 8 relations. We scaled the data to $150K$, $1.5M$, and $6M$ tuples in the Customer, Order, and Lineitem tables respectively.

Policies: For the Census schema we use policies (Person, ϵ) and (Household, ϵ) where the private object is a single individual, or a household, respectively. For the TPC-H schema we used (Customer, ϵ) policy, which protects the presence of customers in the database.

Workload: Summary File 1 (SF-1) [1] is a set of tabulations released by the U.S. Census Bureau. We parsed their description and constructed two workloads of SQL queries: W_1 and W_2. W_1 contains 192 complex queries, most of which contain joins and self joins on the base tables Household and Person as well as correlated sub-queries. An example query is the *"Number of people living in owned houses of size 3 where the householder is a married Hispanic male."* The

Table 3: View Statistics for queries of W_2.

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

second workload $W_2 \supset W_1$ includes an additional 3,493 linear counting queries on Person relation. An example linear query is the *"Number of males between 18 and 21 years old."*. For evaluation of TPC-H we used queries q_1, q_4, q_{13}, q_{16} from the benchmark to derive W_3 a workload of 61 queries, by expanding on the GROUP BY clause of the original queries.

PrivateSQL configuration: The synopsis generation and budget allocation are configurable, as described in Section 8 and listed in Table 2. For the LearnThreshold algorithm described in Section 6.3, we set threshold as $\theta = 0.9$ and budget as $\epsilon_{mf} = 0.05 \cdot \epsilon$.

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

9.2 Overall Error Analysis

We evaluate PrivateSQL on datasets Census_{PM} and Census_{NC} using workloads W_1 and W_2 and both Person and Household. Then we evaluate on TPC-H with the W_3 workload and Customer policy.

Error Rates: Figs. 7 and 8 summarize the RelError distribution of PrivateSQL across different input configurations, stratified by the true query answer sizes. In each figure we draw a horizontal solid black line at $y = 1$, denoting relative error of 100%. A mechanism that always outputs 0 would achieve this error rate.

PrivateSQL achieves low error on the majority of the queries. For the Person policy and Census_{NC} dataset, PrivateSQL achieves at most 2% RelError on 75% of the W_1 queries and at most 6% RelError on 50% of the W_2 queries (Figs. 7a and 7c). For the Household policy (Fig. 7b) all error rates are increased. The noise necessary to hide the presence of a household is much larger as removing one household from the dataset affects multiple rows in the Person table. PrivateSQL also offers high accuracy answers for the W_3 workload on the TPC-H benchmark, with $> 60\%$ of the queries achieving less than 10% relative error (Fig. 7d).

Fig. 8a shows error on the Census_{PM} dataset, using W_1 workload and Person policy. The trends are similar to the Census_{NC} case, but the error is higher as query answers are significantly smaller on Census_{PM} than on Census_{NC}. Fig. 8b shows results on the Census_{NC}, across varying ϵ values. As expected, PrivateSQL incurs smaller error higher values of ϵ. We omit presentation of other configurations due to space constraints.

Error vs Query Size: Naturally, true query sizes affect the relative error rates as much as the query sensitivities; we now examine those effects. In Fig. 7 and Fig. 8a the results are grouped by the size of the true query answer. The number of workload queries in each group is $\{0 - 10^3 : 24, 10^3 - 10^4 : 73, >10^4 : 93\}$ for W_1 and $\{0 - 10^3 : 1869, 10^3 - 10^4 : 811, 10^4 - 10^5 : 742, >10^5 : 253\}$ for W_2. Queries with size $<10^3$ have the highest error. As the true answer size increases, the error drops by an order of magnitude. Under the Person policy, 95% of queries in W_1 and W_2 with size $>10^3$ have error $<10\%$. The median error for queries in W_1 with true answer $>10^4$ is $<.1\%$. This further highlights the real-world utility of PrivateSQL.

View Sensitivities: In Table 3 we show statistics about the views generated from PrivateSQL for workload W_2, dataset Census_{NC}, and both Person and Household policies. Rows of the table correspond to *groups of views* that have the same sensitivity. The second column shows the number of queries that are answerable from views in the group. The rest of the table summarizes the sensitivity of views in each group and the median absolute error (QError) across queries answerable from these views under Person and Household policy, resp. For instance, there are 3575 queries answerable by views with sensitivity 1 under Person policy, and have a median absolute error of 85.

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

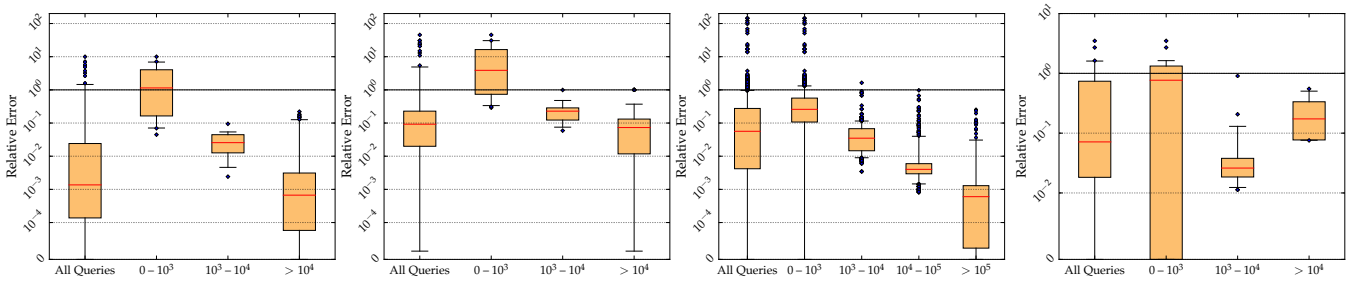
We also derived the equivalent view statistics for TPC-H. For W_3 PrivateSQL creates 4 views with computed sensitivities: $0, 104, 182, 390$ and median QError values: $0, 111$, 112K, 3.5K respectively. Again we see that the sensitivity to error connection is non-linear due to factors like truncation.

9.3 Comparison with Prior Work

We next compare with Flex [24], though a direct comparison is difficult for several reasons. Flex is designed for answering one query at a time, while PrivateSQL answers multiple queries under a common budget. Flex satisfies (ϵ, δ)-differential privacy, a relaxation of DP, whereas for PrivateSQL, $\delta = 0$. PrivateSQL supports multiple privacy policies, while Flex does not (and specifically cannot support the Household policy). We set $\delta = 1/n$ for Flex, where n is the number of rows in the Person table, and consider the Person policy.

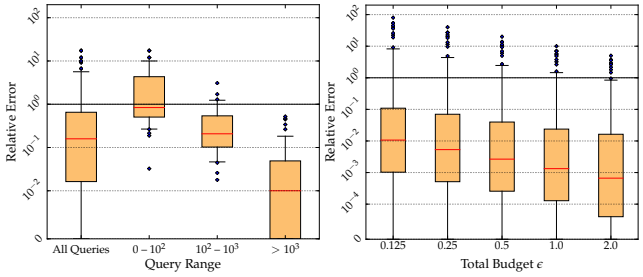
For our first comparison, we compare PrivateSQL against Baseline_{Flex}, a natural extension of Flex adapted for answering a workload of queries, where the privacy budget is evenly divided across the set of answered queries. Then, we provide a more direct "apples to apples" comparison by (a) running both systems one query at a time and (b) comparing their sensitivity engines.

Workload Query Answering We evaluate performance on workloads W_1 and W_2 on Census_{NC} dataset. Flex does not support 42 queries of W_1, which are complex queries containing correlated subqueries. We omit these from the evaluation. In Fig. 9 we present the results, with error distributions again stratified by query size. For the W_1 workload, the Baseline_{Flex} relative error rate exceeds 1 for more than 75% of the queries, while PrivateSQL has error less than 2% for 75% of the queries. Even for large query sizes



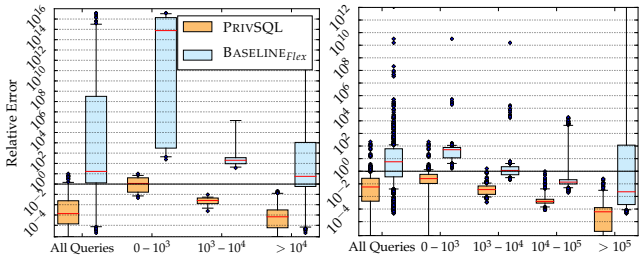
(a) W_1, Person, CENSUS$_{NC}$ (b) W_1, Household, CENSUS$_{NC}$ (c) W_2, Person, CENSUS$_{NC}$ (d) W_3, Customer, TPC-H

Figure 7: Relative error rates of PRIVATESQL. Left is W_1 on the CENSUS$_{NC}$ dataset for Person and Household policies. Right is W_2 on CENSUS$_{NC}$ for Person policy and W_3 on the TPC-H. Error rates stratified by true query answer size.



(a) W_1 on CENSUS$_{PM}$. (b) W_1 on CENSUS$_{NC}$.

Figure 8: Relative error rates for CENSUS$_{PM}$ dataset (left), as well as for different ϵ values (right), both under Person policy.



(a) W_1 Workload (b) W_2 Workload

Figure 9: Relative error comparison between BASELINE$_{Flex}$ and PRIVATESQL for workload answering on the CENSUS$_{NC}$ dataset.

($> 10^4$), BASELINE$_{Flex}$ has high error rates, as W_1 mostly contains complex queries with high sensitivity. For the W_2 workload (Fig. 9b) the trends are similar.

One factor that contributes to PRIVATESQL achieving comparably lower error than the baseline extension of FLEX is that it has more sophisticated support for workloads: VSELECTOR groups together queries which may compose parallely and enjoy a tighter privacy analysis, and techniques like W-NNLS in the synopsis generator use least squares inference to further reduce the error of query answers.

Single Query Answering To provide a more direct comparison with FLEX, we run our system in "single query mode", denoted by PRIVATESQL$_{sqm}$, which takes as input a workload containing a single query and returns a private synopsis to answer that query. We evaluate both systems on workload W_1 on CENSUS$_{NC}$ and Person policy and use a per-query budget of $\epsilon_q = 0.01$. We omit showing results for queries in $W_2 \setminus W_1$ as those queries have the same sensitivity, and hence same error under both systems.

This evaluation allows us to decouple error improvements due to workload-related components – such as VSELECTOR, BUDGETALLOC, and PRIVSYNGEN – and focus on the query analysis components SENSCALC and VREWRITE.

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

Next, we isolate the sensitivity engines of both FLEX and PRIVATESQL and compute only the *sensitivity bounds* (without truncation or smoothing). In Fig. 11 we show our results using the same groups as Fig. 9. For all queries SENSCALC offers a strictly better sensitivity analysis with improvements ranging up to 37× on FLEX supported queries. For group #2 that contains $> 40\%$ of the W_1 queries, SENSCALC offers an improvement of 4×.

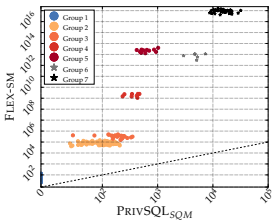


Figure 10: Comparing QERROR rates of W_1 queries on CENSUS$_{NC}$.

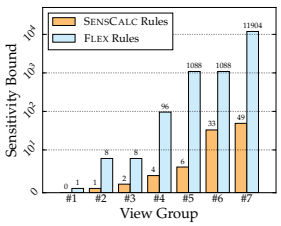
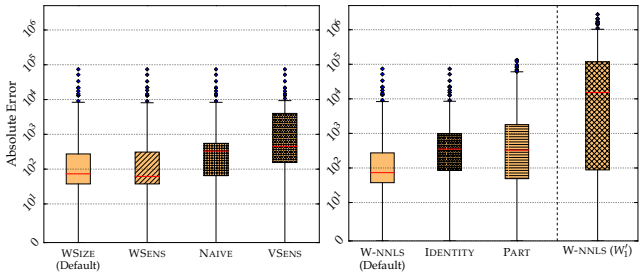


Figure 11: Comparison of FLEX and PRIVATESQL sensitivity

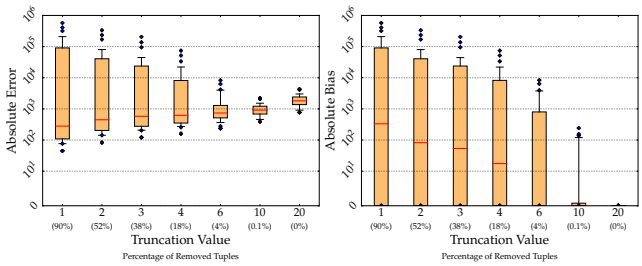
9.4 System Analysis

We next study the effect of alternative choices for varying components of PRIVATESQL. For brevity, we show results on CENSUS$_{NC}$ with the Person policy and workload W_1.



(a) BUDGETALLOC choices. (b) PRIVSYNGEN choices.

Figure 12: Distribution of absolute error for different instantiations of PRIVATESQL (a) shows the effect of BUDGETALLOC and (b) of PRIVSYNGEN.



(a) Absolute error. (b) Absolute bias.

Figure 13: Error and bias distributions of truncation-affected queries, for different truncation values. Numbers in parentheses denote the percentage of tuples truncated at the corresponding value.

Effect of Budget Allocator: In Fig. 12a we show the absolute error distribution of PRIVATESQL for different BUDGETALLOC choices. WSIZE and WSENS offer the best error rates, with comparable performance.

Effect of Synopsis Generator: In Fig. 12b we show the absolute error distribution of PRIVATESQL for different PRIVSYNGEN choices. For representative workload W_1 (left of the dotted line), we see that W-NNLS outperforms the other 2 methods. The non-negative least squares inference technique offers significant advantage since it optimizes for the exact queries that the analyst submits.

Effect of Representative Workload: We create W_1', a smaller representative workload of 35 queries that capture the join structures of queries in W_1 The change in representative workload only affects the W-NNLS synopsis generator, as IDENTITY and PART are workload agnostic (Section 8). The results show that the performance of W-NNLS deteriorates when W_1' is used instead of W_1 (Fig. 12b, right of the dotted line). This suggests that data owners with little knowledge about analyst queries may prefer to instantiate PRIVATESQL with IDENTITY or PART.

Effect of Truncation Operator: The truncation rewrite operation of VREWRITER might introduce bias in the synopses generated – due to tuples being dropped from the base tables. To quantify this bias, we isolate the queries for which Algorithm 2 adds a truncation operator in the query plan of their corresponding view. For all queries in our workloads, the truncated attribute is HID in Person and in PRIVATESQL the LEARNTHRESHOLD as described returns w.h.p. a threshold value of 4. For those queries and for different truncation levels, we measure their total error as well as their bias due to the addition of truncation in their corresponding views. In Fig. 13 we summarize our results.

Small truncation values imply less noise (tighter view sensitivity bounds) but more dropped tuples. For small truncation values, bias dominates overall error. However, note that some queries have 0 bias even for truncation value 1 (e.g., counting households with a single person is not affected by a truncation value of 1). As the truncation value increases, the boxplots narrow but also rise. They narrow because the high error queries improve as their main source of error is bias which drops with increasing truncation value. They rise because increasing the truncation value causing more noise to added to query answers, hurting low error queries. Empirically, we see that a truncation choice between 4 and 6 offers the best of both worlds.

10. CONCLUSIONS

We introduced PRIVATESQL, a first of its kind system that permits differentially private SQL query answering over relational database schemas with key constraints. The system is innovative in several dimensions: (a) it allows a rich set of privacy policies to be expressed, (b) it generates private synopses of views over the base tables to enable answering sets of SQL queries under a common and fixed privacy budget, and (c) it employs semijoin rewriting, truncation, and constraint-oblivious sensitivity analysis to ensure high accuracy. We empirically evaluated its efficacy on real and benchmark workloads of SQL queries.

PRIVATESQL is a first step towards a broader research agenda into differentially private relational databases. PRIVATESQL currently only supports COUNT queries. Handling other aggregate queries is an important research direction. MEDIAN and QUANTILE queries can already be handled by first estimating a CDF (which is a set of counts). To handle SUM and AVG, SENSCALC needs to be extended to keep track of the minimum and maximum values attributes can take, as the range impacts sensitivity. Truncation operators may be needed when attributes are skewed. Another limitation of PRIVATESQL is that query answering is straightforward. Another interesting research direction is to use methods on answering queries using views [18], and statistical relational inference techniques [17] to make query answering from noisy synopses more accurate.

Acknowledgements: This work was supported by the National Science Foundation under grants 1253327, 1408982, 1409143, and 1409125; and by DARPA and SPAWAR under contract N66001-15-C-4067. 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.

11. REFERENCES

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

[2] Census scientific advisory committee fall meeting. https://www.census.gov/about/cac/sac/meetings/2018-12-meeting.html.

[3] Tpc benchmark h. https://http://www.tpc.org/tpch/.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[5] M. Arapinis, D. Figueira, and M. Gaboardi. Sensitivity of counting queries. In *ICALP*, pages 120:1–120:13, 2016.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[7] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar. Privacy, accuracy, and consistency too: A holistic solution to contingency table release. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 273–282. Association for Computing Machinery, Inc., June 2007.

[8] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[9] S. Chen and S. Zhou. Recursive mechanism: Towards node differential privacy and unrestricted joins. In *ACM SIGMOD*, 2013.

[10] W.-Y. Day, N. Li, and M. Lyu. Publishing graph degree distribution with node differential privacy. In *SIGMOD*, 2016.

[11] A. Differential Privacy Team. Learning with privacy at scale, 2017.

[12] X. Ding, X. Zhang, Z. Bao, and H. Jin. Privacy-preserving triangle counting in large graphs. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, CIKM '18, pages 1283–1292, New York, NY, USA, 2018. ACM.

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

[14] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, 2010.

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

[16] Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[17] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[18] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[19] S. Haney, A. Machanavajjhala, J. M. Abowd, M. Graham, M. Kutzbach, and L. Vilhuber. Utility cost of formal privacy for releasing national employer-employee statistics. In *SIGMOD*, 2017.

[20] M. Hardt, K. Ligett, and F. Mcsherry. A simple and practical algorithm for differentially private data release. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2339–2347. Curran Associates, Inc., 2012.

[21] M. Hay, C. Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. In *ICDM*, 2009.

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

[23] X. He, A. Machanavajjhala, and B. Ding. Blowfish privacy: tuning privacy-utility trade-offs using policies. In *ACM SIGMOD*, pages 1447–1458, 2014.

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

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

[26] S. P. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Analyzing graphs with node differential privacy. In *TCC*, 2013.

[27] C. Li, M. Hay, G. Miklau, and Y. Wang. A Data- and Workload-Aware Algorithm for Range Queries Under Differential Privacy. *PVLDB*, 7(5):341–352, 2014.

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

[29] W. Lu, G. Miklau, and V. Gupta. Generating private synthetic databases for untrusted system evaluation. In *ICDE*, 2014.

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

[31] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *PVLDB*, 11(10):1206–1219, 2018.

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

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

[34] W. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *PVLDB*, 6(14):1954–1965, 2013.

[35] W. Qardaji, W. Yang, and N. Li. Priview: Practical differentially private release of marginal contingency tables. In *Proceedings of the 2014 ACM SIGMOD*

International Conference on Management of Data, SIGMOD '14, pages 1435–1446, New York, NY, USA, 2014. ACM.

[36] 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`.

[37] X. Xiao, G. Wang, and J. Gehrke. Differential privacy via wavelet transforms. *IEEE Trans. on Knowl. and Data Eng.*, 23(8):1200–1214, Aug. 2011.

[38] 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.

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

[40] X. Zhang, R. Chen, J. Xu, X. Meng, and Y. Xie. Towards Accurate Histogram Publication under Differential Privacy. *Proc. SIAM SDM Workshop on Data Mining for Medicine and Healthcare*, 2014.