Front.Comput.Sci.
DOI

RESEARCH ARTICLE

SMT-Based Query Tracking for Differentially Private Data Analytics Systems

Chen LUO, Fei HE (✉)

School of Software, Tsinghua University, Beijing 100084, China

Abstract Differential privacy enables analyzing sensitive data in a privacy-preserving manner. In this paper, we focus on the online scenario where each analyst is assigned a *privacy budget* and submits queries interactively. However, existing differentially private data analytics systems like PINQ process each query independently, which may cause the unnecessary waste of the privacy budget. Motivated by this, we present an SMT-based query tracking approach to save the privacy budget usage. Briefly, for each query, our approach automatically locates the past queries which access disjoint parts of the database w.r.t. the current query to save the total privacy cost with the help of SMT solving techniques. We further propose an optimization based on the explicitly specified column ranges to facilitate the search process.

We have implemented a prototype of our approach with Z3, and conducted several sets of experiments. The results show our approach can save considerable amount of budget usage with negligible overhead over query processing.

1 Introduction

Large amount of personal data are being collected, analyzed, and shared by organizations. For example, retail companies analyze the sales data to discover business trends, and hospitals share the medical records to research institutes for research purposes. However, as the data usually contain a lot of private information, protecting the individual's privacy has become a major concern for these organizations. Thus, in the past decades, various privacy definitions and models have been proposed to enable privacy-preserving data analysis.

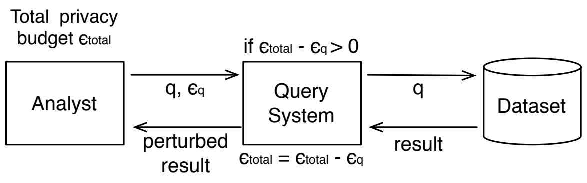

Total privacy budget ϵ_{total}
if ϵ_{total} - ϵ_q > 0
Analyst
q, ϵ_q
perturbed result
Query System
$\epsilon_{total} = \epsilon_{total} - \epsilon_q$
q
result
Dataset

Fig. 1 Workflow of Online Differential Privacy

Differential privacy [1] is one of the strongest privacy models in publishing the statistics of sensitive data. Informally, differential privacy requires the results should be approximately the same in case of removing any individual's record. One common approach to achieve differential privacy is to perturb the result with some well-chosen random noise, which is controlled by the parameter ϵ. Intuitively, larger ϵ requires more noise and in return provides stronger privacy guarantees.

Differential privacy can be offline, where the query batch is available beforehand, or online, where the analyst submits queries interactively. In this paper, we focus on the online setting, which is more applicable in practice. Figure 1 illustrates a typical workflow of an online differentially private data analytics system. Each data analyst is assigned a privacy budget ϵ_{total}. When submitting a query q, the analyst also specifies the desired accuracy ϵ_q. Then, the system checks whether there is enough ϵ_{total} left. If so, the system queries the dataset, subtracts ϵ_q from ϵ_{total}, and answers q with the perturbed result. Otherwise, q is denied.

One major research problem in online differential privacy is to save the privacy budget when answering queries. As [2]

E-mail: hefei@mail.tsinghua.edu.cn

pointed out, when queries access disjoint parts of a dataset, the total privacy cost is the maximum, rather than the sum, of the privacy costs of these queries. However, existing differentially private data analytics systems such as PINQ [2] process each query independently, which may cause the necessary waste of the privacy budget. Thus, in this paper, we present an approach which automatically tracks all submitted queries to save the privacy budget.

Basically, our approach partitions the submitted queries into several subsets such that the queries in each subset are disjoint with each other. Thus, the privacy cost of each subset is the maximum of the privacy costs of the enclosing queries, and the total cost is the sum of the costs of all subsets. However, two challenges need to be addressed for this approach. First, finding an optimal partition to minimize the total cost, which can be viewed as an instance of the set partitioning problem, is NP-hard. To address this, we present an greedy partitioning algorithm which is suitable for online query tracking.

Second, deciding whether two SQL queries access disjoint parts of a database, i.e., disjointness checking, is highly non-trivial. To handle this, we rely on the SMT solving techniques, which are widely used in the software engineering community. Briefly, we encode each query into a logical formula with the property that two queries are disjoint if the conjunction of their logical formulas is unsatisfiable. Moreover, since SMT solving is a relatively expensive operation, we further propose an optimization based on the explicitly specified column ranges. The basic idea is that if we can find a column such the ranges specified by two queries are disjoint, then the queries must be disjoint. But if this process fails, we simply fall back to SMT solving for disjointness checking.

In summary, the main contributions of this paper is summarized as follows:

- We present an SMT encoding scheme to encode SQL queries into logical formulas for disjointness checking;
- With this primitive, we propose an online query set partitioning algorithm for computing total privacy cost;
- We further present an optimization based on the explicitly specified column ranges;
- Finally, extensive experiments show our approach can save considerable amount of budget usage with negligible overhead over query processing.

The rest of the paper is organized as follows. § 2 introduces some background for this paper, and § 3 formalizes the problem definition of this paper. § 4 presents the details of our query tracking approach. § 5 further proposes an optimization based on column ranges. § 6 reports the experimental evaluation of our approach. Finally, § 7 discusses the related works and § 8 concludes this paper.

2 Background

In this section, we review some basic concepts related to this paper.

2.1 Relational Database

The relational database is based on the relational model [4], where the data is organized into one or more relations.

Definition 1 [Relation Schema] A *relation schema* $R(a_1 : D_1, \ldots, a_n : D_n)$ is defined by a relation name R and a set of attributes $a_1, \ldots, a_n$, each of which takes values in the domains $D_1, \ldots, D_n$ respectively.

Definition 2 [Relation Instance] A *relation instance* (or simply, relation) r on a relational schema $R(a_1 : D_1, \ldots, a_n : D_n)$ is a set of tuples $\tau \in D_1 \times \ldots \times D_n$.

Given a relation schema $R(a_1 : D_1, \ldots, a_n : D_n)$, we denote all possible tuples $\tau \in D_1 \times \ldots \times D_n$ of R as $\mathcal{T}_R$, and all relations induced by R as $2^{\mathcal{T}_R}$. Given a tuple τ, we use $\tau(a_i)$ to denote the value of the attribute a_i in τ.

Definition 3 [Relation Database] A *database schema* $\mathcal{S}$ is a set of relation schemas $R_1, \ldots, R_n$. A *database instance* $I_{\mathcal{S}}$ (or simply, database) on $\mathcal{S}$ is a set of relation instances $r_1, \ldots, r_n$, each of which is associated with the relation schemas $R_1, \ldots, R_n \in \mathcal{S}$ respectively.

To query a database, a common approach is to use relational algebra. Relational algebra is defined as a family of algebras operating on relations, and can be viewed as the semantic foundation of SQL. Due to space limitation, we assume some familiarity with relational algebra and SQL, and a reference material can be found in [4].

For the ease of discussion, we make the following conventions. A relation r can either be a *base relation*, i.e., $r \in I_{\mathcal{S}}$, or a *derived relation*, i.e., computed from other relations. We use the term *table* to refer to a base relation, and the term *column* to refer to an attribute in a base relation. Throughout the paper, we use the following database schema as a running example.

Example 1 Consider the following database schema for a small retail company.

- Customer($\underline{\text{c\_id:int}}$, c_name: string, c_age: int, c_zip: string, c_salary: int)
- Item($\underline{\text{i\_id: int}}$, i_name: string, i_price: double)
- Sale($\underline{\text{s\_id: int, si\_id}}$, s_time: timestamp, sc_id: int, s_num: double)

The primary keys have been underlined. The Customer table stores customers' information, and the Item table stores items' information. Finally, sales are stored in the Sale table, which has a composite key of *s_id* and *si_id* (referring to the Item table).

2.2 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a decision problem for deciding the satisfiability of first-order formulas with various theories. Some typical theories include integer, real number and bit vector.

We assume the reader is familiar with the first-order logic, and a reference material can be found in [5]. Informally, the satisfiability problem of a first-order logical formula φ is to find a structure M (including a domain D and an interpretation I) such that φ evaluates to true under M (denoted as $M \models \varphi$). For example, consider the following formula where both x and y are real variables:

$$x > 0 \wedge y > 0 \wedge x + y < 1$$

Here the function + is interpreted as the add operation on real numbers, the predicate > is interpreted as the greater than comparison on real numbers. Note that each variable is assigned with a *sort*, which defines the set of values that the variable can take. For example, both x and y are with real sort. Since x and y are both free variables, we can interpret both x and y as 0.1 (i.e., $I(x) = 0.1$ and $I(y) = 0.1$) such that the formula evaluates to true. Thus, the above formula is satisfiable.

In the following discussion, we assume the domain D is the numerical domain. We further assume the arithmetic functions (e.g., +, -, × and ÷) and predicates (e.g, >, ⩽, <, ⩽, = and ≠) are interpreted as usual conventions. Thus, to check the satisfiability of a logical formula φ, we only need to find an interpretation I which assigns each variable in φ with a value in D such that φ evaluates to *true*.

2.3 Differential Privacy

Differential privacy [1] was first introduced by Dwork in 2006, which informally requires the computations should be approximately the same when any individual's record is removed from the dataset. Thus, differential privacy guarantees that the participation of any individual cannot be observed from the results.

A dataset A is close to another dataset B, denoted as $A \sim B$, if A and B only differ on the addition of at most one record.

Definition 4 [Differential Privacy] We say a query q satisfies ϵ-differential privacy if for any datasets A and B such that $A \sim B$, and any set of possible outputs $S \subseteq Output(q)$,

$$\Pr(q(A) \in S) \leqslant \Pr(q(B) \in S) \times \exp(\epsilon)$$

One notable property of differential privacy is the composability, where multiple queries can be composed sequentially or in parallel [2].

Theorem 1 [Sequential Composition] Let $q_1, \ldots, q_k$ each provides $\epsilon_1, \ldots, \epsilon_k$-differential privacy. The sequence of $q_1(A), \ldots, q_k(A)$ provides $(\sum\limits_i \epsilon_i)$-differential privacy, where A is the dataset.

Theorem 2 [Parallel Composition] Let $q_1, \ldots, q_k$ each provides $\epsilon_1, \ldots, \epsilon_k$-differential privacy and $D_1, \ldots, D_k$ be arbitrary disjoint subsets of the input domain D. The sequence of $q_1(A \cap D_1), \ldots, q_k(A \cap D_k)$ provides $\max\limits_i(\epsilon_i)$-differential privacy, where A is the dataset.

The composability is important for online privacy-preserving data analysis. However, composing queries sequentially with Theorem 1 may quickly use up the privacy budget since the total privacy cost grows linearly w.r.t. the number of queries. One classical approach to leverage the parallel composition theorem is to use the GroupBy or Partition Operator [2], which partitions the dataset into disjoint groups and computes a value for each group. For example, the following query computes the average salary for each zip code and should only consume the budget once:

```
SELECT c_zip, AVG(c_salary)
FROM Customer
GROUP BY c_zip
```

However, this approach requires the analyst with the sense of budget-saving in mind, and sometimes it is difficult to rephrase the queries with the GroupBy operator explicitly. To overcome these limitations, we present our approach in the following sections.

3 Problem Statement

The basic idea of our approach is to track all submitted queries and automatically apply the parallel composition theorem (Theorem 2) to save the privacy budget. Previously, lots of works exist to improve the accuracy of reduce the privacy cost, but most of them only target at certain restricted types of queries, e.g., count queries or linear queries (§ 7). Instead, the approach proposed in this paper is applicable for almost all relational operators and can be viewed as a complement to these works.

Since differentially private queries can only output aggregated values rather than tuples, we assume that each query ends up with one aggregate operation. Note that if a query q has multiple aggregate operations inside, we can simply decompose q into multiple queries and track them separately. We further assume all queries operate on a same database instance $I_{\mathcal{S}} = \{r_1, \ldots, r_n\}$ with the schema $\mathcal{S} = \{R_1, \ldots, R_n\}$. Since a database may contain multiple relations, we define a virtual relation as follows, which constitutes the dataset analyzed by the queries.

Definition 5 [Virtual Relation] Given a database instance $I_{\mathcal{S}} = \{r_1, \ldots, r_n\}$ with schema $\mathcal{S} = \{R_1, \ldots, R_n\}$, the *virtual relation* r_v is defined as $r_v = r_1 \times \ldots \times r_n$ and the schema R_v of r_v is defined as $R_v = R_1 \times \ldots \times R_n$. Denote all possible tuples τ_v of R_v (called *virtual tuples*) as $\mathcal{T}_{R_v} = \mathcal{T}_{R_1} \times \ldots \times \mathcal{T}_{R_n}$.

The central problem of our approach is how to define and check queries access different parts of a database. Intuitively, a virtual tuple τ_v is accessed by a query q if τ_v *could* influence the output of q. Thus, two queries are disjoint if the tuples accessed by them do not overlap. Note that, according to Theorem 2, the disjointness should be defined on all possible database instances, not just the current one.

Definition 6 [Accessed Tuples] Given a query q and a database schema $\mathcal{S}$, let R_v be the virtual relation schema of $\mathcal{S}$. A virtual tuple $\tau_v \in \mathcal{T}_{R_v}$ is *accessed* by q if there exists a database instance $I'_{\mathcal{S}}$ with the corresponding virtual relation r'_v such that $q(r'_v) \neq q(r'_v \cup \{\tau'_v\})$. Denote all virtual tuples accessed by q as $acc(q)$.

Definition 7 [Disjointness] Given queries q_1 and q_2, we say q_1 and q_2 are *disjoint*, denoted as $q_1 \parallel q_2$, if $acc(q_1) \cap acc(q_2) = \emptyset$.

In § 4, we discuss how to check disjointness with SMT solving techniques. We further discuss an optimization for disjointness checking with column ranges in § 5. With the definition of disjointness, we can then compute the total privacy cost of a set of queries Q by partitioning Q as follows.

Definition 8 [Query Set Partition] Given a set of queries Q, the partition P of Q is a set of subsets of Q such that:

- each query $q \in Q$ is included in one and only one subset $S \in P$, and
- for any subset $S \in P$, all queries in S are disjoint with each other, i.e., $\forall S \in P. \forall q_1, q_2 \in S. (q_1 \parallel q_2)$.

After the query set is partitioned, the total privacy cost can be computed based on the following theorem.

Theorem 3 Let a set of queries $Q = \{q_1, \ldots, q_k\}$ each provides $\epsilon_{q_1}, \ldots, \epsilon_{q_k}$-differential privacy and P be a partition of Q, then the set of queries Q provides $\sum\limits_{S \in P} \max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy.

Proof Since we assume that given a database instance $I_{\mathcal{S}}$ with schema $\mathcal{S}$, the actual dataset analyzed by the queries is the virtual relation r_v of $I_{\mathcal{S}}$, the domain of the dataset is all virtual tuples $\mathcal{T}_{R_v}$ of $\mathcal{S}$. For each subset $S \in P$, since all queries in S are disjoint with each other, we can find a partition P_D over $\mathcal{T}_{R_v}$ such that each subset in P_D corresponds to $acc(q)$ of a query $q \in S$. Thus, by applying Theorem 2, the queries in S provide $\max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy. Further applying Theorem 1, we have that all queries in Q provides $\sum\limits_{S \in P} \max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy. □

Note that finding an optimal partition, i.e., minimizing the privacy cost, of a query set can be viewed as an instance of the set partitioning problem, which is NP-hard. To handle this, in § 4 we propose a greedy partitioning algorithm suitable for online tracking. Moreover, we further discuss how to extend this partitioning algorithm with the range optimization in § 5.

4 Query Tracking

In this section, we present the details of our query tracking approach. We first discuss how to encode a query into a logical formula for disjointness checking, and then present a greedy partitioning algorithm for computing total privacy cost.

4.1 Disjointness Checking

As discussed before, the problem of disjointness checking is to determine whether the accessed tuples of two queries are disjoint or not. The basic idea is that we transform each query q into a logical formula φ_q, which characterizes the tuples accessed by q, such that two queries are disjoint if the conjunction of their logical formulas is unsatisfiable. With this property, the disjointness checking problem can be effectively solved with the SMT solving techniques.

We assume the query q has been parsed and optimized into an optimized query plan. To distinguish multiple references of a same relation in a query, we assume each attribute in a query is attached with a unique id based on the relation reference. For simplicity, we only consider attributes with the numerical or string types. For numerical attributes, we support arithmetic operations and comparisons, but only equality/inequality comparisons are supported for the attributes with the string types. Unsupported attributes and operations are simply ignored during the transformation.

Given a query q, we perform the post-order traversal over the query plan of q to compute the formula φ_q in a recursive way. Let p be the current plan operator, we denote the logical formula computed for p as φ_p, the schema of the relation output by p as R_p, and $a_p \in R_p$ as an attribute output by p. For each plan operator p, we also maintain a map $\mathbb{V}_p$, we also maintain a map which maps each attribute output by p to a logical variable or a term. For unary operators, we use the subscript c to denote the components of its child operator, i.e., φ_c, R_c, a_c, and $\mathbb{V}_c$. For binary operators, we use the subscript l and r to denote the components of its left and right operators respectively. Note that we assume q only contains one Aggregate operator at the top, which is ignored in the following discussion since it has no effect on the accessed tuples.

Relation. For the Relation operator, φ_p is simply set as *true* since all tuples in the relation are considered accessed. In the meanwhile, for each supported attribute a_p output by the operator, we create a variable (called *attribute variable*) for $\mathbb{V}_p(a_p)$ with a proper sort as follows:

- for an attribute with the integral type, e.g., int and short, we create a variable with the *integer* sort.
- for an attribute with the fractional type, e.g., double, we create a variable with the *real* sort.
- for an attribute with the string type, we create a variable with the *integer* sort, and each constant string is mapped to an integer.

Projection π. Since π does not filter any tuple, φ_p is simply set as φ_c. For each attribute a_p output by π, we transform the expression defining a_p to a term over attribute variables correspondingly based on $\mathbb{V}_c$. If the transformation succeeds, then $\mathbb{V}_p(a_p)$ is set as the result term, otherwise the attribute a_p is simply ignored. For example, consider the query $\pi_{c\_age/10 \to age\_group, first(c\_name) \to first\_name}(Customer)$. Let v_{c_age} be the variable for c_age. Then $\mathbb{V}_p(age\_group)$ is set as $v_{c\_age}/10$, while $first\_name$ is simply ignored since $first(c\_name)$ is unsupported.

Restriction σ. In contrary to π, σ filters rather than transforms tuples. Thus, $\mathbb{V}_p$ is simply set as $\mathbb{V}_c$, but φ_p is set as $\varphi_c \wedge \varphi_\sigma$, where φ_σ is computed by calling the function $\textsc{Transform}_\sigma$ as follows. Briefly, $\textsc{Transform}_\sigma$ takes as input a condition expression in σ and recursively transforms the expression into a logical formula over attribute variables. For each comparison, $\textsc{Transform}_\sigma$ returns a corresponding predicate if all the operations and attributes are supported, and returns NULL otherwise. For AND, $\textsc{Transform}_\sigma$ returns NULL if all the transformed children are NULL, and returns the conjunction of the transformed children which are not NULL otherwise. For OR, $\textsc{Transform}_\sigma$ returns NULL if one of the transformed children is NULL, and returns the disjunction of the transformed children otherwise. For NOT, $\textsc{Transform}_\sigma$ returns NULL if the transformed child is NULL, and returns the negation of the transformed child otherwise. Finally, if $\textsc{Transform}_\sigma$ returns NULL for the input expression, then φ_σ is simply treated as *true* and ignored.

Join $\bowtie$. Since $\bowtie$ outputs all the attributes output by its child operators, $\mathbb{V}_p$ is set as the union of $\mathbb{V}_l$ and $\mathbb{V}_r$. Let $\varphi_\bowtie$ be a logical formula transformed from the join condition as in σ, φ_p is computed for different join types as follows:

- inner join: $\varphi_l \wedge \varphi_r \wedge \varphi_\bowtie$.
- left outer join: φ_l.
- right outer join: φ_r.
- full outer join: $\varphi_l \vee \varphi_r$.

Intuitively, for inner join, only the joined tuples satisfying the join condition are output by $\bowtie$. While for left outer join, all tuples output by the left operator are output by $\bowtie$ despite of the tuples output by the right operator. Right outer join is dual to left outer join. Finally, for full outer join, all tuples output by the left and the right operators are output by $\bowtie$.

Union $\cup$. For $\cup$, all tuples output by the left and the right operators are output by $\cup$. Moreover, each tuple output by $\cup$ is output by either the left operator or the right operator. Thus, for each supported attribute a_p output by $\cup_p$, we create a fresh variable with the proper sort for $\mathbb{V}_p(a)$. Let $\varphi_\cup$ be $\bigwedge_{a_p}(\mathbb{V}_p(a_p) = \mathbb{V}_l(a_l)) \vee \bigwedge_{a_p}(\mathbb{V}_p(a_p) = \mathbb{V}_r(a_r))$, where for each supported attribute a_p output by $\cup$, a_l and a_r denote the corresponding attributes output by the left and the right operators respectively. Then φ_p is set as $(\varphi_l \vee \varphi_r) \wedge \varphi_\cup$.

Intersect $\cap$. For $\cap$, only the tuples output by both the left and right operators are output by $\cap$. Similarly, for each supported attribute a_p output by $\cap$, we create a fresh variable with the proper sort for $\mathbb{V}_p(a_p)$. Let $\varphi_\cap$ be $\bigwedge_{a_p}(\mathbb{V}_p(a_p) = \mathbb{V}_l(a_l)) \wedge \bigwedge_{a_p}(\mathbb{V}_p(a) = \mathbb{V}_r(a_r))$, where for each attribute a_p output by $\cap$, a_l and a_r denote the corresponding attributes output by the left and right operator respectively. Then, φ_p is set as $\varphi_l \wedge \varphi_r \wedge \varphi_\cap$.

Set-Difference –. – removes the tuples output by the right operator from the tuples output by the left operator. However, the actual removed tuples depend on the given database instance. For soundness, we consider all tuples output by the left operator are output by –. Thus, $\mathbb{V}_p$ is simply set as $\mathbb{V}_l$, and φ_p is set as φ_l.

Other Operators. For other operators including Sort, Distinct and Limit, they neither transform nor filter tuples. Thus, φ_p is simply set as φ_c, and $\mathbb{V}_p$ is set as $\mathbb{V}_c$.

Connecting Attributes with Columns. By far, the computed φ_p cannot be used for disjointness checking directly since φ_p only contain attribute variables, which are not shared among queries. To handle this, we connect φ_p with column variables as the columns the same for all queries. For each query q, we assume a map $\mathbb{R}_q$ which maps each relation schema R to a set of relation references R_q in q. Let $\varphi_c \equiv \bigwedge_{R \in \mathcal{S}}(\bigvee_{R_q \in \mathbb{R}_q(R)}(\bigwedge_{a \in R_q} colvar(a) = v_a))$, where $colvar(a)$ denotes the column variable for an attribute a. Intuitively, for each table referenced by a query (one or more times), the value of column must be equal to the value of an attribute in some referenced relation. Thus, the logical formula φ_q for a query q is computed as $\varphi_p \wedge \varphi_c$.

Example 2 For example, consider the following query q:

```
SELECT SUM(s_num)
FROM Customer INNER JOIN Sale ON c_id = sc_id
WHERE c_age<=30 AND c_zip = '10000'
```

The query plan of q is shown in Figure 2, and each attribute is attached with a unique id (#n). For the Relation operators Customer and Sale, both φ_p are set as *true* while $\mathbb{V}_p$ are initialized by creating a variable for each attribute accordingly. After processing $\bowtie$, φ_p is set as $true \wedge true \wedge v_{c\_id\#1} = v_{sc\_id\#1}$, while $\mathbb{V}_p$ is set as the union of that of both Customer and Sale. After σ, φ_p becomes

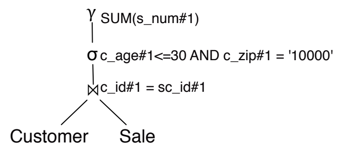


Fig. 2 Example Query Plan

$true \wedge true \wedge v_{c\_id\#1} = v_{sc\_id\#1} \wedge v_{c\_age\#1} \leqslant 30 \wedge v_{c\_zip\#1} = 1$, where we assume '10000' is mapped to 1, and $\mathbb{V}_q$ remains unchanged. Finally, as the query only refers Customer and Sale once, the final formula φ_q is

$$
\begin{aligned}
& v_{c\_id\#1} = v_{sc\_id\#1} \wedge v_{c\_age\#1} \leqslant 30 \wedge v_{c\_zip\#1} = 1 \wedge \\
& (v_{c\_id} = v_{c\_id\#1} \wedge v_{c\_name} = v_{c\_name\#1} \wedge v_{c\_age} = v_{c\_age\#1} \wedge \\
& v_{c\_zip} = v_{c\_zip\#1} \wedge v_{c\_salary} = v_{c\_salary\#1} \wedge v_{s\_id} = v_{s\_id\#1} \wedge \\
& v_{s\_time} = v_{s\_time\#1} \wedge v_{sc\_id} = v_{sc\_id\#1} \wedge v_{si\_id} = v_{si\_id\#1} \wedge \\
& v_{s\_num} = v_{s\_num\#1})
\end{aligned}
$$

The following theorem states the correctness of our transformation algorithm.

Theorem 4 Given a query q and a database schema $\mathcal{S}$, for any accessed tuple $\tau_v \in acc(q)$, there must exist a structure $M = (D, I)$ such that for each column c, $I(v_c) = \tau_v(c)$ and $M \models \varphi_q$.

Proof Sketch. Due to the lengthiness of the complete proof, we only show the sketch here, and the complete proof can be found in the extended version [].

Basically, we show this statement with two steps. First, we use structural induction to over the query plan to show the statement for input tuples, i.e., the tuples in the Cartesian product of all referenced relations for a plan operator. The statement then becomes that given a plan operator p, for any accessed input tuple τ_p^i, there must exist an interpretation I_p such that for each supported attribute a_i in τ_p^i, $I_p(v_{a_i}) = \tau(a_i)$, where v_{a_i} is the variable for a_i, and $I_p(\varphi_p) = true$. Second, we connect accessed input tuples with virtual tuples to conclude the proof.

$\square$

We can then check the disjointness based on the following corollary.

Corollary 1 Given queries q_1 and q_2, let φ_{q_1} and φ_{q_2} be their logical formulas respectively. If $\varphi_{q_1} \wedge \varphi_{q_2}$ is unsatisfiable, then q_1 and q_2 must be disjoint.

Optimization. The number of variables in a logical formula has a direct impact over the time for SMT solving. Here we discuss several optimizations to reduce the number of variables in φ_q. First, when initializing $\mathbb{V}$ for the Relation operator, we can only create variables for the attributes used in the subsequent operators rather than for all attributes. We could apply the same optimization when connecting φ with column variables, i.e., only the variables of the columns which are used in the query are included in φ_c. Second, if a table is only referred once, which is common in practice, we can simply use column variables when traversing the query rather than create attribute variables and connect with column variables later. For example, the optimized φ_q for the example query becomes the following formula, which is much simpler than before:

$$v_{c\_id} = v_{sc\_id} \wedge v_{c\_age} \leqslant 30 \wedge v_{c\_zip} = 1$$

4.2 Greedy Partitioning

With the disjointness checking primitive, we now discuss the partitioning algorithm to compute the total privacy cost. As mentioned in § 3, finding a optimal partition to minimize the total privacy cost is NP-hard. Therefore, we present an greedy partitioning algorithm suitable for online query tracking. The pseudocode of the algorithm is shown in Figure 3. The function TRACK is called for each query q.

Note that the subsets in the partition P are ordered by the number of containing queries, i.e., the subsets with fewer queries come first. For each subset S, we maintain a formula φ_S, which is the disjunction of all formulas of the containing queries. Then for each query q, TRACK checks the k smallest subsets in P (line 3-6), where k is specified by the user. If $\varphi_S \wedge \varphi_q$ is unsatisfiable, which means q is disjoint with all queries in S, then q is added into S (line 5). The function ADD simply adds the query q into the subset S, and updates φ_S and the total privacy cost ϵ_S accordingly(line 9-11). Otherwise, if no suitable subset is found, a new subset is created (line 7). The function CREATE simply creates a subset S which only contains the query q, and the formula φ_S and the privacy cost ϵ_S is set as φ_q and ϵ_q respectively (line 13-16). After the partition is computed, the total privacy cost is the sum of the costs of all subsets, i.e., $\epsilon_{total} = \sum_{S \in P} \epsilon_S$.

Note that the parameter k is necessary for online query tracking since otherwise the time for tracking each query would grow linearly w.r.t. the number of subsets in the partition. The impact of k on balancing the efficiency and the

```
 1: $P \leftarrow \emptyset$                                ▷ $P$ is the partition
 2: function TRACK($q$)
 3:     for the first $k$ subsets $S$ in $P$ do
 4:         if $\varphi_S \wedge \varphi_q$ is unsatisfiable then
 5:             ADD($q, S$)
 6:             return
 7:     CREATE($q$)
 8: function ADD($q, S$)
 9:     $S \leftarrow S \cup \{q\}$
10:     $\varphi_S \leftarrow \varphi_S \vee \varphi_q$
11:     $\epsilon_S \leftarrow$MAX($\epsilon_S, \epsilon_q$)
12: function CREATE($q$)
13:     $S \leftarrow \{q\}$
14:     $\varphi_S \leftarrow \varphi_q$
15:     $\epsilon_S \leftarrow \epsilon_q$
16:     Add $S$ into $P$
```

Fig. 3 Pseudocode for Greedy Partitioning Algorithm

total privacy cost will be discussed in § 6. Obviously, more heuristics are available to further optimize the partitioning algorithm, e.g., the order of subsets and the chosen of a subset in case of multiple suitable candidates. We leave the investigation of possible heuristics as a future work.

5 Range Optimization

Although the query tracking approach discussed in the previous section works in practice, tracking each query still needs k times SMT solving. However, SMT solving is a relatively expensive operation. To reduce the number of SMT solving, we observe that queries often explicitly specify the ranges of some columns. For example, the following query specifies the range of *c_age* should not less than 18.

```
SELECT avg(c_salary) FROM Customer
WHERE c_age>=18 AND ...
```

Thus, for two queries q_1 and q_2, if we can find a column c such that c is accessed by both q_1 and q_2, but the ranges for c accessed by q_1 and q_2 are disjoint, then q_1 and q_2 must be disjoint.

In the remaining of this section, we first discuss how to resolve column ranges from a query for disjointness checking, and then discuss how to adapt the partitioning algorithm with column ranges.

```
1: function RESOLVE($\varphi_p$, $\mathbb{C}_q$)
2:     $\varphi_s \leftarrow$ SIMPLIFY($\varphi_p$)
3:     $\varphi_d \leftarrow$ ToDNF($\varphi_s$)
4:     $\Delta_a \leftarrow NULL$
5:     for $\varphi_{conj}$ in $\varphi_d$ do
6:         $\Delta_t \leftarrow$ RESOLVECONJUNCTION($\varphi_{conj}$)
7:         if $\Delta_a = NULL$ then $\Delta_a \leftarrow \Delta_t$
8:         else $\Delta_a \leftarrow$ MERGE($\Delta_a$, $\Delta_t$)
9:     $\Delta_q \leftarrow$ COLLAPSE($\Delta_a$, $\mathbb{C}_q$)
10:    return $\Delta_q$
11: function MERGE($\Delta_a$, $\Delta_t$)
12:     $\Delta_r \leftarrow \emptyset$
13:     for $v_a$ in $Dom(\Delta_a) \cap Dom(\Delta_t)$ do
14:         $\Delta_r(v_a) \leftarrow$ UNION($\Delta_a(v_a)$, $\Delta_t(v_a)$)
15:     return $\Delta_r$
16: function COLLAPSE($\Delta_a$, $\mathbb{C}_q$)
17:     $\Delta_q \leftarrow \emptyset$
18:     for $v_c$ in $Dom(\mathbb{C}_q)$ do
19:         if $\forall v_a \in \mathbb{C}_q(v_c).\Delta_a(v_a)$ is defined then
20:             $\Delta_q \leftarrow$ UNIONALL($\Delta_a(\mathbb{C}_q(v_c))$)
21:     return $\Delta_q$
```

Fig. 4 Pseudocode for Resolving Column Ranges

5.1 Range Resolution

Previously, we transformed each query q into a logical formula $\varphi_q \equiv \varphi_p \wedge \varphi_c$. We now discuss how to resolve the ranges of the accessed columns (or equivalently, column variables) from φ_q for disjointness checking. Recall that we only support the columns with the numerical or string types. For the numerical column, the range is represented as a set of disjoint intervals. While for the column with the string type, the range is represented as a set of inclusive or exclusive values.

The pseudocode for resolving column ranges is shown in Figure 4. The function RESOLVE takes as input the logical formula φ_p, which is the left part of φ_q, and a map $\mathbb{C}_q$, which maps each column variable to a set of corresponding attribute variables, and outputs a map Δ_q, which maps each column variable to a range.

For simplicity, we only consider binary comparisons of attribute variables with constants (e.g., $>$, $\geqslant$, $<$, $\leqslant$, $=$ and $\neq$). Thus, the formula φ_p is first simplified to remove unsupported predicates (line 2). In SIMPLIFY, unsupported predicates are simply treated as NULL and are then propagated the same way as in TRANSFORM$_\sigma$ in § 4. The simplified φ_s is then rewritten into the disjunctive normal form (DNF) $expr_d$ to extract attribute ranges (line 3).

Δ_a at line 4 is a map which maps each attribute variable to a range, and is initialized as NULL. For each conjunctive clause φ_{conj} in φ_d, RESOLVECONJUNCTION is called to resolve attribute ranges, which is fairly simple since φ_{conj} only contains conjunctions. The ranges resolved from φ_{conj} is stored into a temporary map Δ_t (line 6), which is then merged with Δ_a (line 7 to 8). Since Δ_t and Δ_a are disjunctive, the range of an attribute variable v_a is defined only when both $\Delta_t(v_a)$ and $\Delta_a(v_a)$ are defined and the result range is the union of the both (line 14).

By now, Δ_a only contains attribute ranges. Thus, we then call the function COLLAPSE to compute column ranges. Since a column variable v_c can be equal to any of the attribute variables in $\mathbb{C}_q(v_c)$, its range is then set as the union of the ranges of all the corresponding attribute variables (line 20), where $\Delta_a(\mathbb{C}_q(v_c))$ returns the set of ranges of all the attribute variables in $\mathbb{C}_q(v_c)$.

Example 3 For example, consider the following query q:

```
SELECT AVG(salary) FROM Customer
WHERE c_age ⩾ 20 AND c_age ⩽ 30 AND
      (c_zip = '10000' OR c_zip = '10001')
```

First, q is transformed into a logical formula $\varphi_q \equiv \varphi_p \wedge \varphi_c$, where φ_p is:

$$c\_age\#1 \geqslant 20 \wedge c\_age\#1 \leqslant 30 \wedge$$
$$(c\_zip\#1 =' 10000' \vee c\_zip\#1 =' 10001')$$

For simplicity, we use attribute names to represent attribute variables directly. During the simplification process, φ_p is unchanged since all predicates are supported. Then, φ_p is transformed into the following DNF φ_d:

$$(c\_age\#1 \geqslant 20 \wedge c\_age\#1 \leqslant 30 \wedge c\_zip\#1 =' 10000') \vee$$
$$(c\_age\#1 \geqslant 20 \wedge c\_age\#1 \leqslant 30 \wedge c\_zip\#1 =' 10001')$$

φ_d contains two conjunctive clauses, and calling RESOLVECONJUNCTION gives Δ_1 and Δ_2 as follows:

$$\Delta_1(c\_age\#1) = [20, 30], \Delta_1(c\_zip\#1) = \{10000\}$$
$$\Delta_2(c\_age\#1) = [20, 30], \Delta_2(c\_zip\#1) = \{10001\}$$

By merging Δ_1 and Δ_2, we get Δ_a as follows:

$$\Delta_a(c\_age\#1) = [20, 30], \Delta_a(c\_zip\#1) = \{10000, 10001\}$$

As the last step, since q only refers the Customer table once, the column ranges Δ_q is thus the same as Δ_a.

Finally, the following theorem states the column ranges computed above is conservative.

Theorem 5 Given a query q and the corresponding logical formula φ_q, for any structure $M = (D, I)$ such that $M \models \varphi_q$, let v_c be a variable for a column c, if $\Delta_q(v_c)$ is defined then $I(v_c) \in \Delta_q(c)$.

Proof Sketch. Given a query q, let $\varphi_q = \varphi_p \wedge \varphi_c$ be the corresponding logical formula. From the simplification process, it is straightforward that $\varphi_q \to \varphi_s$. Moreover, since $\varphi_s \leftrightarrow \varphi_d$, we only need to show the statement holds for $\varphi_d \wedge \varphi_s$, which can be easily achieved by considering the process of resolving column ranges from $\varphi_d \wedge \varphi_c$. □

We can further check the disjointness of queries with column ranges based on the following corollary.

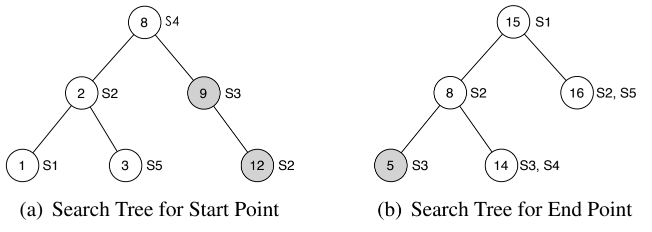
Corollary 2 Given queries q_1 and q_2, and the corresponding column ranges Δ_{q_1} and Δ_{q_2}, if there exists a column c such that both $\Delta_{q_1}(v_c)$ and $\Delta_{q_2}(v_c)$ are defined and $\Delta_{q_1}(v_c)$ is disjoint with $\Delta_{q_2}(v_c)$, then q_1 and q_2 are disjoint.

5.2 Partitioning with Range

Now we discuss how to integrate the partitioning algorithm with column ranges. Recall that the central operation of the partitioning algorithm is to find a target subset S for a query q such that q is disjoint with each query in S. To locate the target subset with column ranges, we attach each subset S with a map Δ_S, where for each column variable v_c, $\Delta_S(v_c)$ is set as the union of $\Delta_q(v_c)$ for each query $q \in S$. Note that if $\Delta_q(v_c)$ for some query $q \in S$ is undefined, then $\Delta_S(v_c)$ is also undefined. Then, from Corollary 2, given a query q and a subset S, if we can find a column c such that both $\Delta_q(v_c)$ and $\Delta_S(v_c)$ are defined and $\Delta_q(v_c)$ is disjoint with $\Delta_S(v_c)$, then q is disjoint with each query in S. If this process fails, we can fall back to SMT solving.

To facilitate the search of target subsets with column ranges, we further build an index for each column. For a column c with the string type, the index simply points to a set of subsets S where $\Delta_S(v_c)$ is defined. We mainly discuss the index for the numerical column.

Recall that the range of a numerical column is essentially a set of disjoint intervals. Thus, for each numerical column c, we maintain two binary search trees, one for the start points of the intervals and another for the end points. Each tree node, i.e., a start (end) point, contains a set of subsets S such that $\Delta_S(v_c)$ contains an interval starting (ending) at that point. With the index, we can locate the target subset S for a query q on the numerical column c as follows. First, for each



(a) Search Tree for Start Point (b) Search Tree for End Point

Fig. 5 Example Index for Numerical Column

interval I in $\Delta_q(v_c)$, we traverse the start (end) point tree to find the nodes which are greater (less) than the end (start) point of I. The subsets contained in these nodes constitute the candidates since each of their ranges on v_c contains an interval which is disjoint with some interval in $\Delta_q(v_c)$. Then, each candidate subset S is checked whether $\Delta_q(v_c)$ is disjoint with $\Delta_S(v_c)$, and if so, S is the target and the process can be stopped.

Example 4 For example, consider the following subsets, and the ranges for the column c are as follows:

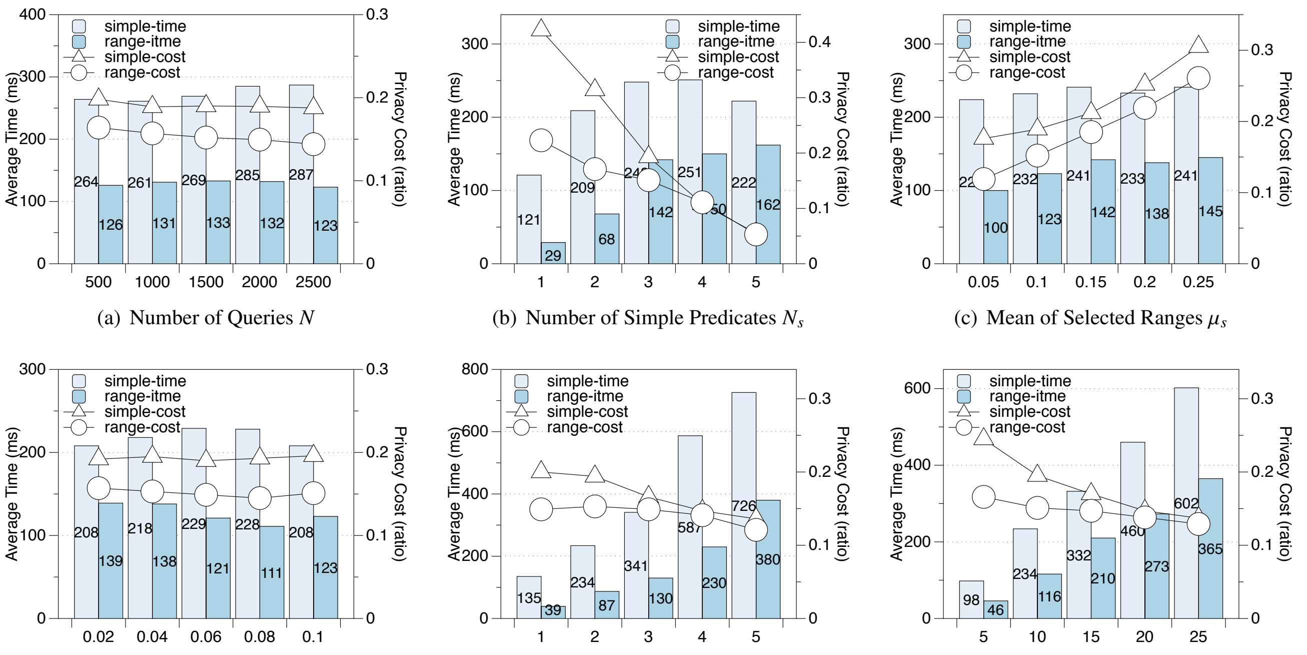
$S_1 : [1, 15]$ $S_2 : [2, 8], [12, 16]$ $S_3 : [1, 5], [9, 14]$
$S_4 : [8, 14]$ $S_5 : [3, 16]$

The index for the column c is shown in Figure 5. Now consider finding a target subset for a query q where the range of c is $[6, 8]$. First, searching start points greater than 8 gives the candidate subsets S_2 and S_3, and searching end points less than 6 gives the candidate subset S_3. After checking the disjointness with q, we realize that only S_3 is the target for q, since the interval $[2, 8]$ in S_2 overlaps with $[6, 8]$ in q.

6 Experimental Evaluation

We have implemented our approach on top of Spark-SQL 1.3.0 [1)] with the Z3 SMT solver [3]. To evaluate our approach, we carried out several sets of experiments. The configurations are as follows. All experiments are performed on a 2.6GHz Intel Core i5CPU with 8GB RAM, and the maximum memory for JVM is set as 4GB. Each experiment is run 3 times, and the average result is reported. We used the Adult table [6] with synthetic queries generated as follows. Each query first filters tuples then computes the aggregation, and consumes the same privacy budget ϵ_q. The filter condition is a conjunction of predicates, including both simple range predicates (e.g., $a > 0$) and complex arithmetic predicates (e.g, $a + b > c$). For simple predicates, the ratio of

1) `http://spark.apache.org/sql/`



(a) Number of Queries N (b) Number of Simple Predicates N_s (c) Mean of Selected Ranges μ_s

(d) Stardard Deviation of Selected Ranges σ_s (e) Number of Complex Predicates N_c (f) Number of Checked Partitions k

Fig. 6 Experimental Results

the selected ranges follows a normal distribution. For complex predicates, we have predefined several templates, such as binary comparisons with add, minus and multiply operations. Each complex predicate is generated by filling the template with randomly chosen columns and constants.

During the experimental evaluation, we mainly focused on the following three metrics: the saving of the privacy budget compared with sequential composition, the running time for tracking each query, and the effectiveness of the range optimization. And we investigated the impact of the following parameters on our approach: the total number of queries N, the number of simple predicates N_s in each query, the mean μ_s and standard deviation (STD) σ_s of the ratio of the selected ranges, the number of complex predicates N_c in each query, and the maximum number of the checked partitions k for each query. The parameter values and the default values (in bold) are shown in Table 1. In each set of experiments, only one parameter is varied while others are set as default.

The experimental results are shown in Figure 6, where the bars denote the average time for tracking each query (in milliseconds), while the lines denote the ratio of the privacy cost of our approach (ϵ_p) of the cost of sequential composition ϵ_s, i.e., ϵ_p/ϵ_s. The time and privacy cost for the original approach in § 4 (denoted as *simple*) and the range-optimized approach in 5 (denoted as *range*) are

Parameter	Values
N	500, **1000**, 1500, 2000, 2500
N_s	1, 2, **3**, 4, 5
μ_s	0.05, **0.1**, 0.15, 0.2, 0.25
σ_s	0.02, **0.04**, 0.06, 0.08, 0.1
N_c	1, **2**, 3, 4, 5
k	5, **10**, 15, 20, 25

Table 1 Experiment Parameters and Values

measured separately.

Figure 6(a) shows the basic performance of our approach under various number of queries. In general, our approach processes each query within milliseconds on average and only consumes about 20% privacy budget compared with sequential composition. The range optimization is also effective, as it reduces the average time for tracking each query by a half. Note that the range optimization also consumes less privacy budget, since all subsets are checked with column ranges despite of the parameter k.

The impact of the number of simple range predicates N_s is shown in Figure 6(b). As one can see, increasing N_s decreases the total privacy cost, since more simple predicates imply that each query accesses smaller ranges, which results in fewer subsets. However, large N_s also increases the time for tracking each query. The reason is that larger N_s causes larger subsets, which in turn make the logical formulas become more complex and also lower the chance of successfully locating target subsets with column ranges.

Figure 6(c) and Figure 6(d) show the impact of the mean μ_s and the standard deviation σ_s of the ratio of the selected ranges respectively. Larger μ_s consumes more privacy budget, since larger ranges selected by each query cause more subsets in the partition. While σ_s has little impact over both the average time and the privacy cost.

Figure 6(e) depicts the impact of the number of complex predicates N_c. As the figure shows, more complex predicates decrease the total privacy cost, since each query accesses a smaller part of the database. In the meanwhile, larger N_c also increases the time for tracking each query since the logical formulas become more complex and, as a result, the time for SMT solving increases.

Finally, Figure 6(f) illustrates the impact of the number of checked partitions k. Increasing k decreases the privacy cost, but the effect become less obvious when k grows larger. In the meanwhile, larger k also increases the time for tracking each query, which grows almost linearly w.r.t. k. From the primary evaluation, 10 to 15 seems a reasonable balance between the privacy cost and the tracking time.

7 Related Work

In this section, we briefly discuss some related works of our work.

SMT Solving. SMT solving techniques have been widely used in the software engineering community, including program verification [9, 10], symbolic execution [11] and software testing [12]. Moreover, many research efforts have also been made to develop efficient SMT solvers. Some classical SMT solvers include Z3 [3], MathSAT [7] and Yices [8]. In this work, we use SMT solving to check the disjointness of queries, which constitutes the primitive of our approach.

Differential Privacy. Differential privacy [1] was first proposed in 2006. Afterwards, numerous algorithms and techniques have been proposed to improve the utility of differential privacy. Here we mainly discuss differential privacy in query processing, including both the offline and online settings.

In the offline setting, the queries are available in advance. Many techniques for different types of queries and applications have been proposed, such as range-count queries [13], histograms [14, 15], set-valued data [16], and high-dimensional data [17] etc. The major difference between these works and our approach is that we do not assume the availability of the queries, and each query is tracked independently.

In the online setting, the system answers the submitted queries interactively. PINQ [2] processes each query independently, and the analyst has to explicitly use the Partition operator for parallel composition. Some other techniques answer a query batch together, but different batches are processed independently. Examples include iReduce [18], matrix mechanism [19], adaptive mechanism [20], and low-rank mechanism [21] etc. Unlike them, Pionner [22] generates an execution plan which reuses the past results for each query to save the privacy cost, which bears some similarity to our work. However, Pioneer is limited to simple linear counting queries, and it is highly non-trivial to extend Pioneer to handle more complex queries and multiple attributes. Instead, we rely on SMT solving techniques to support most relational algebra operators, which is important for practical data analytics systems.

8 Conclusion

In this paper, we present an SMT-based query tracking approach to save the privacy budget for online differentially private data analytics systems. Briefly, we transform the problem of disjointness checking into an instance of SMT solving, and propose an online partitioning algorithm to compute the total cost. We further propose an optimization based on the explicitly specified column ranges in queries. The experimental results show our approach can save considerable amount of budget usage with negligible overhead over query processing.

In the future, we plan to extend our work in the following ways. We plan to investigate effective heuristics for the query partitioning algorithm, especially when the privacy cost of each query varies. We also plan to explore the possibility of reusing past queries which are not totally disjoint with the current query for our approach.

References

1. Dwork C. Differential Privacy. In: International Colloquium on Automata, Languages and Programming. 2006, 1–12

2. McSherry F D. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. Communications of The ACM, 2009, 53: 19–30
3. Moura L M D, Bjørner N. Z3: An Efficient SMT Solver. Z3: An Efficient SMT Solver, 2008
4. Silberschatz A, Korth H F, Sudarshan S. Database system concepts. volume 4. McGraw-Hill New York, 1997
5. Rautenberg W. A Concise Introduction to Mathematical Logic. 2005
6. Lichman M. UCI machine learning repository, 2013
7. Cimatti A, Griggio A, Schaafsma B J, Sebastiani R. The mathsat5 smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 93–107. Springer, 2013
8. Dutertre B. Yices 2.2. In: Biere A, Bloem R, eds, Computer-Aided Verification (CAV'2014). July 2014, 737–744
9. Barnett M, Evan Chang y B, Deline R, Jacobs B, Leino K R M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Boogie: A Modular Reusable Verifier for Object-Oriented Programs, 2005
10. Kroening D, Tautschnig M. Cbmc–c bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems, 389–391. Springer, 2014
11. Godefroid P, Levin M Y, Molnar D A. Automated Whitebox Fuzz Testing. In: Network and Distributed System Security Symposium. 2008
12. Cadar C, Godefroid P, Khurshid S, Păsăreanu C S, Sen K, Tillmann N, Visser W. Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. 2011, 1066–1071
13. Xiao X, Wang G, Gehrke J. Differential privacy via wavelet transforms. Knowledge and Data Engineering, IEEE Transactions on, 2011, 23(8): 1200–1214
14. Hay M, Rastogi V, Miklau G, Suciu D. Boosting the accuracy of differentially private histograms through consistency. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1021–1032
15. Xu J, Zhang Z, Xiao X, Yang Y, Yu G, Winslett M. Differentially private histogram publication. The VLDB JournalThe International Journal on Very Large Data Bases, 2013, 22(6): 797–822
16. Chen R, Mohammed N, Fung B C, Desai B C, Xiong L. Publishing set-valued data via differential privacy. Proceedings of the VLDB Endowment, 2011, 4(11): 1087–1098
17. Zhang J, Cormode G, Procopiuc C M, Srivastava D, Xiao X. Privbayes: Private data release via bayesian networks. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 2014, 1423–1434
18. Xiao X, Bender G, Hay M, Gehrke J. ireduct: Differential privacy with reduced relative errors. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. 2011, 229–240
19. Li C, Hay M, Rastogi V, Miklau G, McGregor A. Optimizing linear counting queries under differential privacy. In: Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 2010, 123–134
20. Li C, Miklau G. An adaptive mechanism for accurate query answering under differential privacy. Proceedings of the VLDB Endowment, 2012, 5(6): 514–525
21. Yuan G, Zhang Z, Winslett M, Xiao X, Yang Y, Hao Z. Low-rank mechanism: optimizing batch queries under differential privacy. Proceedings of the VLDB Endowment, 2012, 5(11): 1352–1363
22. Peng S, Yang Y, Zhang Z, Winslett M, Yu Y. Query optimization for differentially private data management systems. In: Data Engineering (ICDE), 2013 IEEE 29th International Conference on. 2013, 1093–1104