Front.Comput.Sci.
DOI

RESEARCH ARTICLE

SMT-based Query Tracking for Differentially Private Data Analytics Systems

Chen LUO, Fei HE (✉)

Tsinghua National Laboratory for Information Science and Technology (TNList)
Key Laboratory for Information System Security, Ministry of Education
School of Software, Tsinghua University, Beijing 100084, China

Abstract Differential privacy enables sensitive data to be analyzed in a privacy-preserving manner. In this paper, we focus on the online setting where each analyst is assigned a *privacy budget* and queries the data interactively. However, existing differentially private data analytics systems such as PINQ process each query independently, which may cause an unnecessary waste of the privacy budget. Motivated by this, we present a satisfiability modulo theories (SMT)-based query tracking approach to reduce the privacy budget usage. In brief, our approach automatically locates past queries that access disjoint parts of the dataset with respect to the current query to save the privacy cost using the SMT solving techniques. To improve efficiency, we further propose an optimization based on 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 a considerable amount of the privacy budget and each query can be tracked efficiently within milliseconds.

Keywords Differential privacy, privacy budget, satisfiability modulo theory

1 Introduction

Large amounts of personal data are being collected, analyzed, and shared by organizations. For example, a retail company analyzes sales data to discover business trends, and a hospital shares medical records with researchers to promote the study of certain diseases. However, as business data often contains customers' private information, protecting the individual's privacy from being leaked in data analysis has become a major concern of these organizations. In the past few decades, various privacy definitions and models have been proposed to enable privacy-preserving data analysis.

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

Differential privacy [1] is one of the strongest privacy models for publishing the statistics of sensitive data. Informally, differential privacy guarantees that the participation of any individual cannot be observed from the statistics. It requires the statistics to be approximately the same in case that any individual's record is removed from the dataset. One common approach to achieve differential privacy is to perturb the statistics with some well-chosen random noise, which is controlled by the parameter ϵ. Intuitively, a smaller ϵ requires more noise, i.e., less-accurate results, 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. Fig. 1 illustrates the typical workflow of an online differentially private data analytics system. Each data analyst is first 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 sufficient ϵ_{total} remains. 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

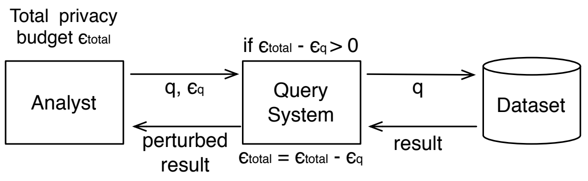


Fig. 1 Workflow of online differential privacy

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

In a nutshell, our approach partitions the submitted queries into subsets such that the queries in each subset access disjoint parts of the dataset 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, there are two major challenges for tracking queries effectively. 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 a greedy partitioning algorithm that is suitable for online query tracking.

Secondly, 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 satisfiability modulo theories (SMT) solving techniques, which have been 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, because SMT solving is a relatively expensive operation, we further propose an optimization using 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, the queries must be disjoint. If this process fails, we simply fall back on SMT solving for disjointness checking.

The main contributions of this paper are summarized as follows:

- we introduce an SMT encoding scheme to check the disjointness of Structured Query Language (SQL) queries;
- with this primitive, we present a greedy partitioning algorithm for online query tracking;
- we further propose an optimization technique using the explicitly specified column ranges;
- finally, extensive experiments show our approach can considerably reduce budget usage with each query being tracked efficiently within milliseconds.

The rest of the paper is organized as follows. § 2 introduces some preliminaries related to this paper, and § 3 formalizes the problem definition. § 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 Preliminaries

In this section, we review some preliminaries related to this paper.

2.1 Relational Database

The relational database is based on the relational model [3], where the data is organized as 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 domains $D_1, \ldots, D_n$, respectively.*

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

Given a relation schema $R(a_1 : D_1, \ldots, a_n : D_n)$, we denote all possible tuples $\tau \in D_1 \times \cdots \times D_n$ 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 (Relational 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 a database) on a database schema $\mathcal{S}$ is a set of relation instances $r_1, \ldots, r_n$, each of which is defined on relation schemas $R_1, \ldots, R_n \in \mathcal{S}$, respectively.*

A common approach to query a database is to use relational algebra. Relational algebra is a family of algebras on relations. Each relational algebra operator takes as input one or two relations, and produces a relation as output. Some operator may also have an extra parameter that specifies the property of that operator. Since relational algebra is viewed as the semantic foundation of SQL, a relational algebra expression is often called as the *query plan* of an SQL query.

In the following, we briefly discuss some common relational algebra operators. More complete reference material can be found in [3].

- Project $\pi_L(r)$: L is an attribute list, whereas r is the input relation. Here π_L outputs a relation that only contains the attributes defined in L.
- Restriction $\sigma_\phi(r)$: ϕ is a logical formula and r is the input relation. Here σ_ϕ filters the tuples of r and only outputs the tuples that satisfy the formula ϕ.
- Join $l \bowtie_\phi r$: ϕ is a logical formula and l and r are two input relations. Here $\bowtie_\phi$ joins l and r by first computing the Cartesian product of l and r, and then filtering the result using the join condition ϕ. There are several types of the join operator, where the above is called *inner join*. *Left outer join* takes an extra step by adding all tuples of l that are filtered by the join condition into the result. *Right outer join* is dual to left outer join and adds the tuples of r. Finally, *full outer join* combines left and right outer joins, and adds all tuples of l and r that are filtered by the join condition into the result.
- Grouping $\gamma_{L_g,L_a}(r)$: L_g and L_a are two attribute lists that contain a set of grouping attributes and aggregate attributes, respectively, and r is the input relation. Here γ_{L_g,L_a} first partitions the tuples of r into groups, where each group contains all tuples having the same values of the grouping attributes in L_g. Then, for each group, it produces one tuple that contains the values of the grouping attributes as well as the aggregations over all tuples of that group.
- Sort $\tau_L(r)$: L is an attribute list and r is the input relation. Here τ sorts the tuples of r using the attributes in L.
- Distinct $\delta(r)$: r is the input relation. Here δ eliminates the duplicate tuples of r.
- Binary operators: Binary operators include $\cup$, $\cap$, and $\setminus$, which calculate the union, intersection, and difference of the two input relations, respectively.

For ease of discussion, we make the following conventions through the paper. A relation r can either be a *base relation*, i.e., $r \in I_S$, or a *derived relation*, i.e., computed from other relations. We use the terms *table*, *column*, and *row* to refer to a base relation, an attribute of a base relation, and a tuple of a base relation, respectively. Moreover, as a query may refer a relation multiple times, we assume each referred attribute in a query has a unique id based on its relation reference.

Example 1. *As a running example, consider the following database schema for a retail company.*

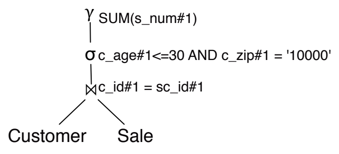


Fig. 2 Example query plan

- *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($\underline{\text{s\_id: int, si\_id: int}}$, sc_id: int, s_time: timestamp, s_num: double)*

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

An example SQL query q_1 is shown as follows:

```
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_1 is shown in Fig. 2, where each attribute is attached with a unique id (#n) based on its relation reference. Intuitively, q_1 first joins the Customer and Sale relations with the condition c_id = sc_id, and only keeps the tuples which satisfy the condition c_age $\leq$ 30 and c_zip $='$ 10000$'$. Finally, it computes the sum of s_sum for all remaining tuples, where all tuples are treated as one group.

2.2 Differential Privacy

Differential privacy [1] was first proposed by Dwork in 2006, and has become the de facto standard of privacy-preserving data analysis. Intuitively, differential privacy requires the computations over a dataset should be approximately the same when any individual's record is removed. In doing so, it protects the individual's privacy by ensuring that the participation of any individual in the dataset cannot be observed from the results.

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

Definition 4 (Differential privacy)**.** *A query function $q : \mathcal{D} \rightarrow R$ satisfies ϵ-differential privacy if for any datasets $A, B \in \mathcal{D}$*

such that $A \sim B$, and any set of possible outputs $S \subseteq R$,

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

From above definition, differential privacy is controlled by the parameter ϵ. Intuitively, a smaller ϵ means less-accurate results and, thus, provides stronger privacy protection. To achieve differential privacy, one classical approach is the Laplace mechanism, which adds some well-chosen random noise to the query result based on its sensitivity.

Definition 5 (Sensitivity)**.** *Given a query function $q : \mathcal{D} \rightarrow R$, the sensitivity of q is*

$$\Delta q = \max_{A,B \in \mathcal{D}.A \sim B} |q(A) - q(B)|$$

Theorem 1 (Laplace mechanism)**.** *Given a query function $q : \mathcal{D} \rightarrow R$ and a dataset $A \in \mathcal{D}$, the Laplace mechanism $M_q(A) = q(A) + Laplace(\sigma)$ provides $(\Delta q/\sigma)$-differential privacy, where $Laplace(\sigma)$ is the Laplace distribution with the mean being 0 and the variance being $2\sigma^2$.*

However, note that our approach is agnostic to the underlying mechanism that implements differential privacy. Thus, we simply assume that the query has been properly perturbed by the Laplace mechanism or other mechanism such as the exponential mechanism [4]. Instead, we are more interested in how multiple queries are composed together, namely sequential composition or parallel composition [2].

Theorem 2 (Sequential Composition)**.** *Let $q_1, \ldots, q_k$ each provide $\epsilon_1, \ldots, \epsilon_k$-differential privacy. Given a dataset A, the sequence of $q_1(A), \ldots, q_k(A)$ provides $(\sum_i \epsilon_i)$-differential privacy.*

Theorem 3 (Parallel composition)**.** *Let $q_1, \ldots, q_k$ each provide $\epsilon_1, \ldots, \epsilon_k$-differential privacy and $D_1, \ldots, D_k$ be arbitrary disjoint subsets of the input domain D. Given a dataset A, the sequence of $q_1(A \cap D_1), \ldots, q_k(A \cap D_k)$ provides $\max_i(\epsilon_i)$-differential privacy.*

Composability is important for online privacy-preserving data analysis. However, composing queries sequentially using Theorem 2 may quickly use up the privacy budget since the total privacy cost grows linearly with respect to 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 an analyst with a sense of budget-saving in mind, and sometimes it is difficult to rephrase queries using the GroupBy operator explicitly. Instead, our approach overcomes these limitations by automatically leveraging parallel composition for submitted queries to save the privacy budget.

2.3 Satisfiability Modulo Theories

The SMT problem is concerned with the satisfiability of first-order logic formulas. First-order logic is defined over a set of logical connectives, variables, functions, and predicates. Constants can be simply treated as 0-ary functions. Typical logical connectives include negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), existential quantification ($\exists$), and universal quantification ($\forall$). Each variable has a *sort*, i.e., domain, that specifies a set of values the variable can take. An *interpretation I* of a first-order formula specifies the meanings of functions and predicates and assigns a value for each variable. A first-order formula φ is *satisfiable* if there exists an interpretation I such that φ evaluates to *true*, denoted as $I(\varphi) = true$.

Note that the satisfiability problem of first-order logic formulas is, in general, undecidable. Thus, the SMT problem is to decide the satisfiability of a given first-order formula φ in the context of some background theory $\mathcal{T}$. A theory $\mathcal{T}$ fixes the sort of variables and the interpretations of functions and predicates. For example, the integer theory supports variables with the integer sort and normal mathematical operations such as add, minus, equality, and inequality. The real number theory is similar to the integer theory, but considers real-valued variables. Moreover, as various theories are often not used in isolation, modern SMT solvers also support the combination of multiple theories.

Thus, given a first-order formula φ, the SMT problem essentially tries to find an interpretation I that assigns a proper value for each variable such that $I(\varphi)$ evaluates to *true*. For example, consider the following formula where both x and y are real variables:

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

Clearly, we can find an interpretation I where $I(x) = 0.1$ and $I(y) = 0.1$ such that the formula evaluates to *true*. However, the above formula becomes unsatisfiable if both x and y are integer variables.

In the remainder of this paper, we only consider numerical theories, i.e., the integer theory and the real number theory.

3 Problem Definition

The basic idea of our approach is to track all submitted queries and automatically apply the parallel composition theorem (Theorem 3) to save the privacy budget. Previously, lots of works have been proposed to improve the utility of differential privacy, but most of them only target 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, i.e., statistics, rather than tuples, we assume that each query ends up with one aggregate operation to output an aggregated value. Note that if a query q has multiple aggregate operations, we can simply decompose q into multiple queries and track them separately. We further assume all queries operate on the same database instance $I_{\mathcal{S}} = \{r_1, \ldots, r_n\}$ on 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 6 (Virtual relation)**.** *Given a database instance $I_{\mathcal{S}} = \{r_1, \ldots, r_n\}$ on the schema $\mathcal{S} = \{R_1, \ldots, R_n\}$, the* virtual relation *r_v is defined as $r_v = r_1 \times \ldots \times r_n$ on the* virtual relation schema *$R_v = R_1 \times \cdots \times R_n$. We denote all possible tuples τ_v of R_v (called* virtual tuples*) as $\mathcal{T}_{R_v} = \mathcal{T}_{R_1} \times \cdots \times \mathcal{T}_{R_n}$.*

The central problem of our approach is to define and check whether queries access disjoint parts of a database. Intuitively, a virtual tuple τ_v is accessed by a query q if τ_v *may* influence the output of q. Thus, two queries are disjoint if the tuples accessed by them do not overlap. Note that according to Theorem 3, disjointness should be defined on all possible database instances, not just the current one.

Definition 7 (Accessed tuples)**.** *Given a query q on 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 $q(r'_v) \neq q(r'_v \cup \{\tau_v\})$. We denote all virtual tuples accessed by q as $acc(q)$.*

Definition 8 (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 using the SMT solving techniques. We further discuss an optimization for disjointness checking using explicitly specified 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 9 (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 4. *Let a set of queries $Q = \{q_1, \ldots, q_k\}$ each provide $\epsilon_{q_1}, \ldots, \epsilon_{q_k}$-differential privacy and P be a partition of Q, then the queries in Q provide $\sum\limits_{S \in P} \max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy.*

Proof. Since we assume that for a database instance $I_{\mathcal{S}}$ on the schema $\mathcal{S}$, the actual dataset analyzed by the queries is the virtual relation r_v of $I_{\mathcal{S}}$, the domain of the analyzed dataset is then the set of 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 3, the queries in S provide $\max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy. Then by applying Theorem 2, all queries in Q provide $\sum\limits_{S \in P} \max\limits_{q_i \in S}(\epsilon_{q_i})$-differential privacy. □

However, finding an optimal partition of a query set to minimize the privacy cost 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. We further discuss how to extend this partitioning algorithm with column ranges in § 5.

Finally, note that the approach proposed in this paper offers the same level of privacy protection as differential privacy does, i.e., the participation of any individual in the dataset cannot be observed from the results. However, our goal is to save the privacy budget when answering differentially private queries. Or, equivalently, answering more queries under a given privacy budget ϵ_{total}.

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 an online partitioning algorithm for computing the total privacy cost.

4.1 Disjointness Checking

As mentioned before, the disjointness checking problem is to determine whether the accessed tuples of two queries are disjoint. The basic idea of our approach is to transform each query q into a logical formula φ_q that characterizes the accessed tuples 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 help of existing SMT solvers, such as Z3 [5].

4.1.1 Query Encoding

Now we discuss how to encode a query q into a logical formula for disjointness checking. We assume the query q has been parsed into a query plan, i.e., a relational algebra expression. For simplicity, we only consider attributes with the numerical or string types. For numerical attributes, we support normal arithmetic operations and comparisons, but only equality/inequality comparisons for string typed attributes. Unsupported attributes and operations are simply ignored during the transformation.

Briefly, we perform post-order traversal over the query plan of the query q to compute the formula φ_q compositionally. Let p be the current plan operator, we denote φ_p the logical formula computed for p, R_p the schema of the relation outputted by p, and $a_p \in R_p$ an attribute outputted by p. For each plan operator p, we also maintain a map $\mathbb{V}_p$, which maps each attribute outputted by p to a variable or a logical term. For unary operators, we use the subscript c to denote the above elements of its child operator, i.e., φ_c, R_c, a_c, and $\mathbb{V}_c$, respectively. For binary operators, we use the subscripts l and r to denote the components of its left and right operators, respectively. Recall that we assume the query q outputs an aggregated value with only one GroupBy operator at the top. Since the GroupBy operator simply aggregates tuples and has no effect on the accessed tuples, it is simply ignored in the following discussion.

Relation. The Relation operator simply refers a table, i.e., a base relation, in the database. Thus, φ_p is simply set as *true* since all tuples are considered accessed. Meanwhile, for each supported attribute a_p in the referred relation, we create a variable (called *attribute variable*) for a_p, which is stored as $\mathbb{V}_p(a_p)$, 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., float and 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_L. Since π_L does not filter any tuple, φ_p is simply set as φ_c. Meanwhile, for each attribute a_p defined in L, we transform the expression defining a_p to a term 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 an attribute list $L_1 = c\_age\#1/10 \rightarrow age\_group\#1, first(c\_name\#1) \rightarrow first\_name\#1$ and let $\mathbb{V}_c(c\_age\#1) = v_{c\_age\#1}$. Then $\mathbb{V}_p(age\_group\#1)$ is set as $v_{c\_age\#1}/10$, whereas $first\_name\#1$ is simply ignored since $first(c\_name\#1)$ is unsupported.

Restriction σ_ϕ. In contrast to π_L, σ_ϕ filters rather than transforming 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 returned by the function $\textsc{Transform}_\sigma$ as follows. Briefly, $\textsc{Transform}_\sigma$ takes as input the filter condition ϕ in σ_ϕ and recursively transforms ϕ into a logical formula based on $\mathbb{V}_c$. For each binary 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 that 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 filter condition ϕ, then φ_σ is simply treated as *true* and ignored.

Join $\bowtie_\phi$. Since $\bowtie_\phi$ outputs a relation that contains all the attributes outputted 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 ϕ as in σ_ϕ, then 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: $\varphi_l \lor \varphi_r$.

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

Union $\cup$**.** For $\cup$, all tuples outputted by the left and right operators are outputted by $\cup$. Moreover, each tuple outputted by $\cup$ is either from the left operator or the right operator. Thus, for each supported attribute a_p outputted by $\cup$, we create a new variable with the proper sort and store it in $\mathbb{V}_p(a_p)$. Let $\varphi_l^{\cup}$ be $\bigwedge_{a_p}(\mathbb{V}_p(a_p) = \mathbb{V}_l(a_l))$, and $\varphi_r^{\cup}$ be $\bigwedge_{a_p}(\mathbb{V}_p(a_p) = \mathbb{V}_r(a_r))$, where for each supported attribute a_p outputted by $\cup$, a_l and a_r denote the corresponding attributes outputted by the left and right operators, respectively. Then, φ_p is set as $(\varphi_l \land \varphi_l^{\cup}) \lor (\varphi_r \land \varphi_l^{\cup})$.

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

Set-Difference $\backslash$**.** $\backslash$ removes the tuples outputted by the right operator from the tuples outputted by the left operator. However, the actual removed tuples depend on the given database instance. For soundness, we consider all tuples outputted by the left operator are outputted by $\backslash$. 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. Recall that we assume each attribute referred in the query has a unique id to differentiate multiple references of the same relation. Thus, the computed formula 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 in the database schema are the same for all queries. For each query q, let $\mathbb{R}_q$ be a map that 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 (without id) for an attribute a. Intuitively, for each table referred by a query, each row in the table must come from one of the referred relations. Finally, the logical formula φ_q for the query q is computed as $\varphi_p \land \varphi_c$.

Example 2. *Consider the query q in Example 1, we show how to encode q into a logical formula φ_q as follows. First, for the Relation operators Customer and Sale, φ_p is always set as true whereas $\mathbb{V}_p$ is initialized by creating a variable for each attribute accordingly. After $\bowtie_{c\_id\#1=sc\_id\#1}$, φ_p is set as $\land v_{c\_id\#1} = v_{sc\_id\#1}$, while $\mathbb{V}_p$ is set as the union of that of both Customer and Sale. After $\sigma_{c\_age\#1 \leq 30\ AND\ c\_zip\#1='1000'}$, φ_p becomes $\land v_{c\_id\#1} = v_{sc\_id\#1} \land v_{c\_age\#1} \leq 30 \land v_{c\_zip\#1} = 1$, where we assume '10000' is mapped to 1, and $\mathbb{V}_q$ remains unchanged. Note that the final operator $\gamma_{(SUM(s\_num\#1))}$ is simply ignored since it has no effect on the accessed tuples. Finally, as the query only refers Customer and Sale once, the final formula φ_q is*

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

Optimization. The number of variables in a logical formula has a direct impact on the performance of SMT solving. Here we discuss several optimizations to reduce the number of variables in the encoded formula φ_q. First, when initializing $\mathbb{V}_p$ for the Relation operator, we only need to create variables for those attributes that are used in the subsequent operators. A similar optimization can be applied when we connect φ_p with column variables, that is, we only need to include the column variables where the columns are used in the query. Second, if a table is only referred once, which is common in practice, we can use column variables directly in $\mathbb{V}_p$ during the post-traversal process. For example, the optimized formula φ_q for the example query is shown as follows, which is much simpler than before:

$$
v_{c\_id} = v_{sc\_id} \land v_{c\_age} \leq 30 \land v_{c\_zip} = 1
$$

4.1.2 Correctness

In this subsection, we show the correctness of our query encoding mechanism.

Theorem 5. *Given a query q on a database schema $\mathcal{S}$, for any accessed tuple $\tau_v \in acc(q)$, there exists an interpretation I such that for each column c, $I(v_c) = \tau(c)$ and $I(\varphi_q) = true$.*

Proof. Before proving this statement, we first introduce some notation. For each plan operator p of a query q, we use $\mathcal{R}^i_p$ to denote a set of base relation schemas (with unique attribute ids) referred by p. We further use R^i_p to denote the Cartesian product of the schemas in $\mathcal{R}^i_p$ and $\mathcal{T}^i_p$ to denote the set of all possible tuples τ^i_p induced by R^i_p. For clarity, the tuple τ^i_p is called the *input tuple* of the plan operator p. Definition 7 can be directly extended to the tuples in $\mathcal{T}^i_p$. Given an input tuple τ^i_p for a plan operator p, we use τ_p to denote the corresponding tuple outputted by p.

Given two interpretations I_1 and I_2 such that I_1 and I_2 are compatible, i.e., for each variable v, $I_1(v) = I_2(v)$, we define the union of I_1 and I_2 as follows, where for each variable v:

$$(I_1 \cup I_2)(v) = \begin{cases} I_1(v), \text{if } I_1(v) \text{ is defined} \\ I_2(v), \text{if } I_2(v) \text{ is defined} \\ \text{undefined, otherwise} \end{cases}$$

Given an interpretation I and a variable v, if $I(v)$ is not defined, we assume v can take any possible value in its sort.

We then show this statement with two steps. First, given a plan operator p, for any accessed input tuple τ^i_p, we show there exists an interpretation I_p such that for each supported attribute a_i in τ^i_p, ${}_pI(v_{a_i}) = \tau(a_i)$ where v_{a_i} is the variable for a_i and $I_p(\varphi_p) = true$. Second, we link input tuples with virtual tuples by connecting attributes to columns to conclude the proof.

For the first statement, we show it by structural induction over the query plan.

Inductive basis: for the Relation operator, the statement trivially holds since $\varphi_p \equiv true$.

Inductive step: for each unary or binary operator, we assume the statement holds for the child operator(s), and prove the statement holds for the current operator.

For the Projection operator π_L, since $\varphi_p \equiv \varphi_c$, the statement trivially holds from the inductive assumption.

For the Restriction operator σ_ϕ, $\varphi_p \equiv \varphi_c \wedge \varphi_\sigma$. From the inductive assumption, we have $I_c(\varphi_c) = true$. From the construction of φ_σ and $\mathbb{V}_p$, it is straightforward that φ_σ is an over-approximation of the filter condition ϕ in σ_ϕ. That is, if the filter condition ϕ is satisfied, then $I_c(\varphi_\sigma)$ must evaluate to *true*. Since τ^i_p is accessed by σ_ϕ, we have that τ_c must not be filtered by σ_ϕ, which means the filter condition ϕ is satisfied. Thus, we have $I_c(\varphi_c \wedge \varphi_\sigma) = true$, and the statement holds.

For the Join operator $\bowtie_\phi$, let $\tau^i_p = \tau^i_l \times \tau^i_r$ and $I_p = I_l \cup I_r$. We consider each join type separately. For inner join, $\varphi_p \equiv \varphi_l \wedge \varphi_r \wedge \varphi_\bowtie$. Since τ^i_p is accessed by inner join, we have that τ_p must be outputted by $\bowtie_\phi$, which implies that τ_l and τ_r are outputted by the left and right operators, respectively, Thus, τ^i_l and τ^i_r are accessed by the left and right operators, respectively. Moreover, we also have that the join condition ϕ is satisfied. Then from the inductive assumption, we have $I_p(\varphi_l \wedge \varphi_r \wedge \varphi_\bowtie) = true$, and the statement holds.

For left outer join, if τ^i_p is accessed, then τ_l must be outputted by left outer join, which means that τ^i_l is accessed by the left operator. Thus, we have $I_p(\varphi_l) = true$, and the statement holds. Right outer join is handled dually. Finally, for full outer join, if τ^i_p is accessed, then either τ_l or τ_r is outputted by the operator, which means that either τ^i_l is accessed by the left operator or τ^i_r is accessed by the right operator. Thus, we have $I_p(\varphi_l \vee \varphi_r) = true$, and the statement holds.

For the Union operator $\cup$, $\varphi_p \equiv (\varphi_l \vee \varphi^\cup_l) \wedge (\varphi_r \vee \varphi^\cup_r)$. Let $\tau^i_p = \tau^i_l \times \tau^i_r$ and $I_p = I_l \cup I_r$. If τ^i_p is accessed by $\cup$, then at least one of τ_l and τ_r is outputted by $\cup$, which means that either τ^i_l is accessed by the left operator or τ^i_r is accessed by the right operator. Thus, from the inductive assumption, we have either $I_p(\varphi_l) = true$ or $I_p(\varphi_r) = true$. Without loss of generality, assume $I_p(\varphi_l) = true$. Then $I_p(\varphi^l_\cup) = true$ trivially holds, since we can simply set $I(\mathbb{V}_p(a_p))$ for each attribute a_p outputted by $\cup$ as $I(\mathbb{V}_l(a_l))$ or $I(\mathbb{V}_r(a_r))$, where a_l or a_r is the corresponding attribute outputted by the left or right operator, respectively. Thus, the statement holds.

For the Intersect operator $\cap$, $\varphi_p \equiv (\varphi_l \wedge \varphi^\cap_l) \wedge (\varphi_r \wedge \varphi^\cap_r)$. Let $\tau^i_p = \tau^i_l \times \tau^i_r$ and $I_p = I_l \cup I_r$. Similarly, if τ^i_p is accessed by $\cap$, which means τ_p is outputted by $\cap$, then τ_l and τ_r are outputted by the left and right operators, respectively. Thus, τ^i_l is accessed by the left operator and τ^i_r is accessed by the right operator. From the inductive assumption, we have $I_p(\varphi_l) = true$ and $I_p(\varphi_r) = true$. Moreover, τ_l and τ_p must be the same, i.e., for each pair of attributes a_l and a_r, $\tau_l(a_l) = \tau_l(a_r)$. Thus, we have $I_p(\varphi^\cap_l \wedge \varphi^\cap_r) = true$, and the statement holds.

Finally, for the Set Difference operator $\setminus$, $\varphi_p \equiv \varphi_l$. Let $\tau^i_p = \tau^i_l \times \tau^i_r$ and $I_p = I_l \cup I_r$. If τ^i_p is accessed by $\setminus$, then τ_l must be outputted by the left operator, which means τ^i_l is accessed by the left operator. Thus, from the inductive assumption, we have $I_p(\varphi_l) = true$, and the statement holds.

We then conclude the proof by connecting input tuples with virtual tuples as follows. Given a query q, let p be the final operator of q. Let $\mathbb{R}_q$ be a map that maps each relation schema to a set of relation references in q. For each input tuple τ^i_p, we define $virtual(\tau^i_p)$ as a set of virtual tuples τ_v such that for each relation schema R in $\mathcal{S}$:

- $\forall c \in R.\tau_v(c) \in D_c$, where D_c is the domain of the column c, if $\mathbb{R}_q(R)$ is undefined;
- $\forall c \in R.\tau_v(c) = \tau^i_p(R_q(c))$, for some $R_q \in \mathbb{R}_q(R)$.

Here we use $R_q(c)$ to denote the attribute in R_q for a given column c. Intuitively, if some relation is not referred in q, the columns in that relation can take any values in the virtual tuple; otherwise, we flatten the multiple references of a relation into multiple virtual tuples. Thus, each relation is referred exactly once in the virtual tuple. Moreover, it is straightforward that for any accessed virtual tuple $\tau_v \in acc(q)$, there must exist an accessed input tuple τ^i_p such that $\tau_v \in virtual(\tau^i_p)$.

Now consider $\varphi_q \equiv \varphi_p \wedge \varphi_c$. Given an accessed virtual tuple τ_v and an accessed input tuple τ^i_p such that $\tau_v \in virtual(\tau^i_p)$, let $I_v = I_p$. From the previous discussion, we have $I_v(\varphi_p) = true$. Then consider $\varphi_c \equiv \bigwedge_{R\in\mathcal{S}} (\bigvee_{R_q\in\mathbb{R}_q(R)} (\bigwedge_{a\in R_q} colvar(a) = v_a))$. For each relation schema R and a set of relation references $\mathbb{R}_q(R)$, let $R_q \in \mathbb{R}_q(R)$ be the relation reference used in constructing τ_v. Then for each attribute $a \in R_q$, we simply set $I_v(colvar(a)) = I_v(v_a)$. Thus, it is straightforward that $I_v(\varphi_c) = true$, and the theorem holds. □

Based on the above theorem, it is straightforward to have the following corollary, which allows us to check the disjointness using SMT solvers.

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 are disjoint.*

4.2 Greedy Partitioning

With the disjointness checking operation, we now present the partitioning algorithm for computing the total privacy cost of submitted queries. As mentioned in § 3, finding an optimal partition to minimize the total privacy cost is NP-hard. Thus, we present a greedy partitioning algorithm suitable for online query tracking. The pseudocode is shown in Fig. 3, where the function Track is called for each submitted query q.

The subsets in the partition P are maintained in ascending order with respect to the size, i.e., smaller subsets come first. For each subset S, we maintain a formula φ_S and a cost ϵ. φ_S is the disjunction of the formulas of all enclosing queries of S. Here ϵ_S is the maximum of the privacy costs of the queries of S. Then for the submitted query q, the function Track checks the first k smallest subsets in P (lines 3–6), where k is specified by the user. For each subset S, if $\varphi_S \wedge \varphi_q$

```
 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 the greedy partitioning algorithm

is unsatisfiable, which means q is disjoint with all queries in S, then S is the target subset of q. Thus, q is added into S (line 5) by calling the function Add. The function Add simply adds the query q into the subset S, and updates the formula φ_S and the total privacy cost ϵ_S accordingly (lines 9–11). Otherwise, if no suitable subset is found, a new subset is created (line 7). The function Create simply creates a subset S that only contains the query q, and the formula φ_S and the privacy cost ϵ_S are initialized as φ_q and ϵ_q, respectively (lines 13–16). After the partition is updated, 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 with respect to the number of subsets. Intuitively, the parameter k balances the efficiency and the total privacy cost of our approach. Smaller k means each query can be tracked more efficiently since each query requires at most k times SMT solving. Meanwhile, it also lowers the chances of locating a target subset, which in turn would create more subsets and, according to Theorem 4, increase the total privacy cost. We experimentally evaluate the effect of k further in § 6.

Obviously, more heuristics are available to further optimize the partitioning algorithm, e.g., the order of subsets and the choice of a subset in the case of multiple suitable candidates. We leave the investigation of possible heuristics as a future work.

5 Range Optimization

Although the query tracking approach presented in the previous section works in practice, tracking each query still needs k times SMT solving, which is a relatively expensive operation. To further 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 be no less than 18:

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

Thus, for any two queries q_1 and q_2, if we can find a column c such that the ranges of c specified by q_1 and q_2 are disjoint, then q_1 and q_2 must be disjoint.

In the remainder 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 columns with the numerical or string types. For the numerical column, the range is represented as a set of disjoint intervals. For the string typed column, the range is represented as a set of inclusive or exclusive values.

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

For simplicity, we only consider binary comparisons of attribute variables with constants (e.g., $>$, $\geq$, $<$, $\leq$, $=$, 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 then propagated in the same way as in TRANSFORM$_\sigma$ in § 4. The simplified φ_s is then rewritten into the disjunctive normal form (DNF) φ_d to extract attribute ranges (line 3). Briefly, φ_d is a disjunction of a set of conjunctive clauses, while each conjunctive clause is a conjunction of optionally negated predicates.

In line 4, Δ_a is a map that maps each attribute variable to a range, and is initialized as NULL. For each conjunctive clause φ_{conj} in φ_d, we call the function RESOLVECONJUNCTION 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 (lines 7–8). 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 stores attribute ranges. We then call the function COLLAPSE to compute column ranges. Since a column variable v_c is equal to one 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 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

$$v_{c\_age\#1} \geq 20 \wedge v_{c\_age\#1} \leq 30 \wedge$$
$$(v_{c\_zip\#1} = '10000' \vee v_{c\_zip\#1} = '10001')$$

During the simplification process, φ_p is unchanged since all predicates are supported. Then, φ_p is transformed into the *following DNF φ_d:*

$$(v_{c\_age\#1} \geq 20 \wedge v_{c\_age\#1} \leq 30 \wedge v_{c\_zip\#1} =' 10000') \vee$$
$$(v_{c\_age\#1} \geq 20 \wedge v_{c\_age\#1} \leq 30 \wedge v_{c\_zip\#1} =' 10001')$$

Here φ_d contains two conjunctive clauses, and calling ResolveConjunction *gives Δ_1 and Δ_2 as follows:*

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

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

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

As the last step, since q only refers the Customer table once, the map Δ_q is the same as Δ_a:

$$\Delta_q(v_{c\_age}) = [20, 30], \Delta_q(v_{c\_zip}) = \{10000, 10001\}$$

Finally, the following theorem states the conservativeness of the resolved column ranges.

Theorem 6. *Given a query q, let φ_q and Δ_q be the corresponding logical formula and column ranges, respectively. For any column variable v_c and any interpretation I such that $I(\varphi_q) = true$, if $\Delta_q(v_c)$ is defined, then $I(v_c) \in \Delta_q(c)$.*

Proof. For the query q, let $\varphi_q \equiv \varphi_p \wedge \varphi_c$. From the function Simplification, it is straightforward that $\varphi_p \rightarrow \varphi_s$. Moreover, since the DNF φ_d is equivalent to φ_s, we have $\varphi_p \rightarrow \varphi_d$. Thus, we only need to show for any interpretation I such that $I(\varphi_d \wedge \varphi_c) = true$, if $\Delta_q(v_c)$ is defined, then $I(v_c) \in \Delta_q(v_c)$.

Let I be an interpretation such that $I(\varphi_d \wedge \varphi_c) = true$, which implies both $I(\varphi_d) = true$ and $I(\varphi_c) = true$. Now consider the resolution of attribute ranges Δ_a from φ_d. Let v_a be an attribute variable such that $\Delta_a(v_a)$ is defined. Since φ_d is the disjunction of a set of conjunctive clauses, there must exist a conjunctive clause φ_{conj} such that $I(\varphi_{conj}) = true$. Let Δ_t be the attribute ranges resolved from φ_{conj} by calling ResolveConjunction. Since $\Delta_a(v_a)$ is defined, we have that $\Delta_t(v_a)$ is also defined. Then from ResolveConjunction, it is straightforward that $I(v_a) \in \Delta_t(v_a)$. Moreover, since $\Delta_a(v_a)$ is the union of $\Delta_t(v_a)$ of all conjunctive clauses in φ_d, we have $I(v_a) \in \Delta_a(v_a)$.

Finally, consider merging the ranges of all attributes for a column c. Let v_c be a column variable such that $\Delta_q(v_c)$ is defined, which implies that for all $v_a \in \mathbb{C}_q(v_c)$, $\Delta_a(v_a)$ is also defined. From $I(\varphi_c) = true$ and the structure of φ_c, we have $I(v_c) = I(v_a)$ for some $v_a \in \mathbb{C}_q(v_a)$, which implies $I(v_c) \in \Delta_a(v_a)$. Moreover, since $\Delta_q(v_c)$ is the union of $\Delta_a(v_a)$ of all $v_a \in \mathbb{C}_q(v_c)$, we have $I(v_c) \in \Delta_q(v_c)$. Thus, the theorem holds. □

We can further check query disjointness using 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 Greedy Partitioning with Range

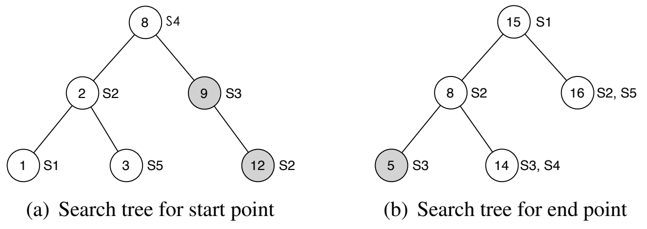
We now discuss how to integrate the resolved column ranges into the partitioning algorithm. Recall that the essential operation of the partitioning algorithm is to find a target subset S for a query q such that q is disjoint with each query in S. To locate the target subset with column ranges, we attach each subset S with a map Δ_S. For each column variable v_c, $\Delta_S(v_c)$ is set as the union of $\Delta_q(v_c)$ of each query $q \in S$. Note that if $\Delta_q(v_c)$ is undefined for some query $q \in S$, then $\Delta_S(v_c)$ is undefined as well. Then from Corollary 2, given a query q, if we can find a subset S and 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 must be disjoint with each query in S. If this fails, we simply fall back to SMT solving to locate target subsets.

To speed up the search of target subsets with column ranges, we further build an index for each column. For a string typed column c, the index simply points to a set of subsets S where $\Delta_S(v_c)$ is defined. For a numerical column, the index is slightly complicated and will be discussed below.

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 this index, we can locate the target subset S for a query q using a numerical column c as follows. First, for each 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 S contained in these nodes constitute the candidates since $\Delta_S(v_c)$ contains an interval that is disjoint with an interval in $\Delta_q(v_c)$. Each candidate subset S is then checked whether $\Delta_q(v_c)$ is disjoint with $\Delta_S(v_c)$. If so, S is the target and the process can be stopped.

Example 4. *For example, consider the following subsets where the ranges for a column c are as follows:*

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



(a) Search tree for start point (b) Search tree for end point

Fig. 5 Example index for a numerical column

The index for the column c is shown in Fig. 5. Now suppose we want to find 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)] using Z3 4.4.2 [5] as the SMT solver. To evaluate our approach, we carried out several sets of experiments. All experiments are performed on a Macbook Pro with 2.7 GHz Intel Core i7 CPU and 16 GB RAM. The maximum memory of JVM is set as 4 GB. Each experiment is performed three times and the average result is reported. For Z3, we use its default configurations and each set of experiments shares one solver instance to enable incremental solving.

In all experiments, we use the Census-Income dataset [6] and synthetic queries. The Census-Income dataset contains 40 attributes and 299285 records. Note that our approach handles queries in an instance-independent manner, which means it only uses the attributes while ignores the underling records. The synthetic queries used in the experiments are generated as follows. Each query first filters tuples and then computes an aggregated value, 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 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 then generated by filling the template with randomly chosen columns and constants.

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

During the experimental evaluation, we mainly focused on the following three metrics: the relative privacy cost, i.e., the privacy cost of our approach compared with the cost of PINQ, the average time of tracking each query, and the effectiveness of the range optimization. Since PINQ processes each query independently, the relative privacy cost is computed as the number of created subsets divided by the total number of queries. Thus, the value of the privacy cost ϵ_q of each query is irrelevant in the experiments. We have 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 (SD) 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 subsets k for each query. The parameter values are listed in Table 1, where the default values are in bold. In each set of experiments, we only vary one parameter whereas the others are set as default.

Table 1 Experiment parameters and values

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

The experimental results are given in Table 2. For the base approach discussed in § 4, we list the relative privacy cost with respect to PINQ and the average time of tracking each query (track time). For the range-optimized approach presented in § 5, we further list the average time of locating subsets using column ranges (range time) and SMT solving (SMT time), and the percentage of queries where a subset is successfully located by column ranges (range hits).

In the first set of experiments, we vary the number of queries N to evaluate the basic performance of our approach. In general, the base approach tracks each query within 100 ms and only consumes 35% of the privacy budget compared with the cost of PINQ. The range-optimized approach only takes 30 ms to track each query and consumes 20% of the privacy budget, which implies the range optimization is effective. For the time cost, the range optimization only incurs less than 1 ms extra overhead for each query (range time), but can successfully locate a subset for more than 70% of queries and, thus, greatly reduces the tracking time. However, when this fails, the time of SMT solving is in general longer than the time of the base approach. The reason is that the logical formulas of the remaining queries are more complex, and require more time for the solver to decide their satisfiability. Meanwhile, since the range optimization checks all subsets using column ranges despite of the parameter k, it also reduces the total privacy cost.

In the second set of experiments, we vary the number of simple range predicates N_s in each query. As one can see, both the privacy cost and the tracking time of our approach are sensitive to N_s. Intuitively, larger N_s means each query accesses a smaller part of the dataset, which in turn reduces the number of created subsets and, thus, the relative privacy cost. Meanwhile, it also increases the number of queries in each subset, which makes the logical formula more complex and increases the time of SMT solving. The results also show the privacy cost of the range-optimized approach is less sensitive to N_s, since it checks all subsets using column ranges. However, larger N_s stills increases its average tracking time. The reason is that besides making the logical formulas more complex, larger N_s also lowers the chances of locating subsets using column ranges since each subset contains more queries.

The next two sets of experiments evaluate the impact of the mean μ_s and the SD σ_s of the selected ranges, respectively. The impact of μ_s on the relative privacy cost is contrary to the impact of N_s, since larger μ_s implies each query accesses a larger part of the dataset. However, although μ_s has little impact on the tracking time of the base approach, the tracking time of the range-optimized approach increases with larger μ_s. The reason is that larger μ_s, i.e., larger column ranges of each query, lowers the chances of finding target subsets with column ranges. In contrast, σ_s has little impact on the privacy cost and the average tracking time of both approaches.

In the fifth set of experiments, we vary the number of complex predicates N_c in each query. As the results show, larger N_c slightly decreases the relative privacy cost, since each query accesses a smaller part of the dataset. Meanwhile, more complex predicates also increase the time for tracking each query since the logical become more complex, especially for non-linear operations, and thus the time of SMT solving increases. However, note that we simply choose the default Z3 configurations in all experiments. It is worth considering optimizing the configurations to improve the performance of our approach under more complex predicates.

Finally, in the last set of experiments, we evaluate the impact of the number of checked subsets k. Increasing k reduces the privacy cost of the base approach, but the effect becomes less obvious when k grows larger. However, k only has little impact over the privacy cost of the range-optimized approach, since it already checks all subsets using column ranges. Meanwhile, the tracking time grows almost linearly with respect to k. From the primary evaluation, 15 to 20 seems to be a reasonable balance between the privacy cost and the tracking time.

7 Related Work

In this section, we briefly discuss some related works.

SMT Solving. SMT solving techniques have been widely used in the software engineering community, such as program verification [7, 8], symbolic execution [9], and software testing [10]. Moreover, many research efforts have also been made to develop efficient SMT solvers. Some classical SMT solvers include Z3 [5], MathSAT [11], and Yices [12]. In this work, we use SMT solving to check the disjointness of queries, which constitutes the primitive operation of our approach.

Differential Privacy. Differential privacy [1] was first proposed in 2006. Here we mainly discuss differential privacy in query processing. In the offline setting, 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]. Unlike these, Pioneer [22] generates an execution plan that 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 count queries, and it is 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.

Query Auditing. Query auditing determines whether queries access some forbidden tuples, which can be checked in instance-dependent [23, 24] or instance-independent [25, 26] manners. Instance-independent query auditing can be viewed as a counterpart of our approach by treating the forbidden tuples as a special query. However, our approach still differs from them in several aspects. First, we consider the definition of disjointness based on virtual tuples for parallel composition. Moreover, most query auditing techniques only consider a limited class of queries, e.g., conjunctive queries or SPJ queries, while we support most relational algebra operators using SMT solving techniques. Finally, disjointness checking is only the basic building block of our approach, and we further partition the query set to compute the total cost.

8 Conclusion

In this paper, we have presented an SMT-based query tracking approach to reduce the privacy cost for online differentially private data analytics systems. In brief, we have transformed the disjointness checking problem into an instance of SMT solving, and present an online partitioning algorithm for tracking queries. We have further proposed an optimization based on the explicitly specified column ranges in queries. The experimental results show our approach can considerably reduce privacy budget usage and each query can be tracked efficiently within milliseconds.

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

ACKNOWLEDGMENTS

This work was supported in part by the Chinese National 973 Plan (2010CB328003), the NSF of China (61672310, 61272001, 60903030, 91218302), and the Chinese National Key Technology R&D Program (SQ2012BAJY4052).

References

1. Dwork C. Differential privacy. In: Proceedings of 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. Silberschatz A, Korth H F, Sudarshan S. Database system concepts. volume 4. McGraw-Hill New York, 1997
4. McSherry F, Talwar K. Mechanism design via differential privacy. In: Proceedings of IEEE Symposium on Foundations of Computer Science. 2007, 94–103
5. De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008, 337–340
6. Lichman M. UCI machine learning repository, 2013
7. Barnett M, Chang B Y E, DeLine R, Jacobs B, Leino K R M. Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of International Conference on Formal Methods for Components and Objects. 2006, 364–387
8. Kroening D, Tautschnig M. CBMC–C bounded model checker. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2014, 389–391
9. Godefroid P, Levin M Y, Molnar D A. Automated Whitebox Fuzz Testing. In: Proceedings of Network and Distributed System Security Symposium. 2008, 151–166
10. 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 International Conference on Software Engineering. 2011, 1066–1071
11. Cimatti A, Griggio A, Schaafsma B J, Sebastiani R. The mathsat5 SMT solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2013, 93–107
12. Dutertre B. Yices 2.2. In: Proceedings of International Conference on Computer Aided Verification. 2014, 737–744
13. Xiao X, Wang G, Gehrke J. Differential privacy via wavelet transforms. IEEE Transactions on Knowledge and Data Engineering, 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. Differentially private histogram publication. In: Proceedings of IEEE International Conference on Data Engineering. 2012, 32–43
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 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 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 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: Proceedings of IEEE International Conference on Data Engineering. 2013, 1093–1104
23. Agrawal R, Bayardo R, Faloutsos C, Kiernan J, Rantzau R, Srikant R. Auditing compliance with a hippocratic database. In: Proceedings of International Conference on Very Large Data Bases. 2004, 516–527
24. Kaushik R, Ramamurthy R. Efficient auditing for complex SQL queries. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2011, 697–708
25. Miklau G, Suciu D. A formal analysis of information disclosure in data exchange. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2004, 575–586
26. Motwani R, Nabar S U, Thomas D. Auditing SQL queries. In: Proceedings of IEEE International Conference on Data Engineering. 2008, 287–296

Chen Luo received his BS degree from Tongji University in 2013 and his Master's degree from Tsinghua University in 2016. He is currently a PhD student in University of California Irvine, USA. His research interests include formal methods and database systems.

Fei He received his BS degree from the National University of Defense Technology in 2002, and the PhD degree from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include satisfiability, model checking, compositional reasoning, and their applications to embedded systems.

Table 2 Experimental results

Experiment	Value	Base Approach		Range-Optimized Approach				
		Track Time	Relative Privacy Cost	Range Time	Range Hits	SMT Time	Track Time	Relative Privacy Cost
Number of Queries N	500	60.00 ms	34.47%	0.30 ms	71.80%	107.31 ms	30.59 ms	21.67%
	1000	57.50 ms	34.37%	0.25 ms	76.23%	93.84 ms	22.35 ms	19.00%
	1500	59.37 ms	35.80%	0.27 ms	77.69%	116.58 ms	26.15 ms	17.67%
	2000	60.60 ms	35.38%	0.26 ms	78.40%	105.41 ms	22.83 ms	17.23%
	2500	62.28 ms	35.53%	0.26 ms	78.79%	96.09 ms	20.45 ms	17.05%
Number of Simple Predicates N_s	1	59.41 ms	77.60%	0.23 ms	71.00%	63.33 ms	18.60 ms	28.60%
	2	54.85 ms	47.27%	0.27 ms	77.67%	77.66 ms	17.58 ms	21.00%
	3	59.53 ms	35.30%	0.28 ms	78.27%	101.74 ms	22.39 ms	18.23%
	4	66.28 ms	28.50%	0.27 ms	68.90%	117.64 ms	36.52 ms	17.83%
	5	71.55 ms	21.43%	0.24 ms	55.93%	105.21 ms	46.08 ms	16.60%
Mean of Selected Ranges μ_s	0.05	56.84 ms	32.57%	0.30 ms	81.83%	105.35 ms	19.19 ms	14.70%
	0.1	61.33 ms	35.50%	0.28 ms	75.73%	103.21 ms	25.12 ms	18.97%
	0.15	62.26 ms	40.07%	0.27 ms	73.27%	98.13 ms	26.30 ms	22.87%
	0.2	66.93 ms	43.93%	0.27 ms	68.97%	101.57 ms	31.57 ms	27.03%
	0.25	61.33 ms	46.60%	0.27 ms	64.30%	94.27 ms	33.86 ms	31.97%
SD of Selected Ranges σ_s	0.02	63.31 ms	36.37%	0.27 ms	76.67%	117.00 ms	22.41 ms	18.73%
	0.04	60.67 ms	35.53%	0.28 ms	76.30%	103.17 ms	24.47 ms	18.80%
	0.06	57.39 ms	35.13%	0.28 ms	78.17%	105.22 ms	23.24 ms	18.57%
	0.08	57.48 ms	36.07%	0.28 ms	78.07%	104.50 ms	23.02 ms	18.27%
	0.1	61.24 ms	36.60%	0.32 ms	77.40%	96.38 ms	22.03 ms	18.67%
Number of Complex Predicates N_c	1	40.96 ms	36.97%	0.28 ms	76.83%	61.54 ms	14.52 ms	19.20%
	2	59.25 ms	35.50%	0.27 ms	77.00%	108.98 ms	25.28 ms	18.67%
	3	91.00 ms	36.53%	0.27 ms	74.77%	163.03 ms	41.14 ms	18.90%
	4	133.68 ms	34.33%	0.27 ms	71.37%	223.12 ms	63.43 ms	18.63%
	5	190.13 ms	32.70%	0.25 ms	67.77%	321.24 ms	101.54 ms	18.53%
Number of Checked Partitions k	5	29.10 ms	44.97%	0.28 ms	78.77%	43.38 ms	9.46 ms	18.57%
	10	58.69 ms	35.47%	0.25 ms	77.23%	109.16 ms	24.99 ms	18.40%
	15	82.84 ms	31.53%	0.25 ms	74.70%	144.21 ms	36.58 ms	18.63%
	20	118.42 ms	27.07%	0.29 ms	74.57%	220.27 ms	56.20 ms	18.30%
	25	165.99 ms	24.47%	0.30 ms	72.60%	307.91 ms	84.25 ms	18.13%