Query Rewriting-Based View Generation for Efficient Multi-Relation Multi-Query with Differential Privacy

Xinglin Du [†1], Peng Tang [†2*], Rui Chen [‡], Ning Wang [§], Chengyu Hu [†3], Shanqing Guo [†4]

[†] School of Cyber Science and Technology, Shandong University, Qingdao, China
[†]Quan Cheng Laboratory, Jinan, China
[†]State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China
[†]xinglin[1]@mail.sdu.edu.cn, {tangpeng[2], hcy[3], guoshanqing[4]}@sdu.edu.cn
[‡]College of Computer Science and Technology, Harbin Engineering University, Harbin, China
[‡]ruichen@hrbeu.edu.cn
[§]Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, China
[§]wangninggzu@gmail.com

ABSTRACT

This research addresses the challenge of conducting multi-query operations across multiple relations while preserving differential privacy. In multi-relation multi-query scenarios, a common strategy is to generate private synopses from multiple views on the base relations. These synopses are carefully adjusted to minimize error rates on a representative query workload, thus reducing any further privacy loss. However, when dealing with queries involving multiple relations, there is often a significant number of nested and derived table queries. Directly generating views for such queries leads to the problem of excessive view proliferation. Thus, there still remains a significant privacy cost issue. To overcome this challenge, we propose a view generation approach based on query rewriting. This approach involves performing equivalent rewriting on nested and derived table queries, ensuring that the number of views does not increase with changes in conditions within subqueries. Throughout the rewriting process, we skillfully address the challenges of maintaining query equivalence before and after rewriting, eliminating individual query operations, and ensuring compatibility across various database platforms. We conducted extensive experiments on real datasets to evaluate our proposed solution. The results demonstrate that our approach provides desirable data utility. The source code and data have been made available at https://github.com/xinglindu/ViewRewrite.

1 INTRODUCTION

Database querying is essential in various applications as it enables efficient data retrieval. However, protecting sensitive individual records from unauthorized disclosures necessitates the integration of differential privacy [7–9, 24, 25] into the querying process. Differential privacy has emerged as the standard for private data release due to its robust protection of individual information. Recently, we have observed that several organizations, including Microsoft [2], Google [1], and the United States Census Bureau [14], have performed multiple practical deployments.

In differential privacy, the privacy budget ε is allocated to restrict privacy loss, and a calibrated amount of noise is added to query results to guarantee ε-differential privacy. However, in real-world scenarios, databases are frequently accessed, and queries often involve multiple relations, posing challenges for differentially private protection. In particular, for a set of queries, to guarantee differential privacy, we can divide the ε into several partitions, and then allocate them to each query. However, the injected noise scale is inversely proportional to the privacy budget, and as the number of queries increases, the privacy budget allocated to each query decreases. This will result in a decrease in the accuracy of the query results. Therefore, the goal of this research is to decrease the privacy budget consumption of all queries across multiple relations. To achieve this goal, a viable strategy [20] is to generate private synopses from multiple views over the base relations. However, further research has revealed that for nested and derived table queries, views often vary with changes in their subquery's filter criteria. Directly generating views for such queries would result in an excessive number of views, making it impractical to reduce privacy budget consumption.

To address this issue, we propose a novel method for generating views based on query rewriting. The core of query rewriting is to decompose, perform algebraic transformations, and merge the filtering criteria of subqueries, to avoid the increase in the number of generated views caused by changes in the filtering criteria. During the query rewriting process, several considerations must be taken into account: 1) **Query Equivalence.** Preserving query equivalence is crucial to avoid scenarios of non-equivalent query rewriting. 2) **Differential privacy constraints.** Constraints imposed by differential privacy scenarios must be fully considered to avoid queries targeting individual records, which has been a challenging problem for publishing views involving non-distinct values, maximum or minimum values. 3) **Database compatibility.** Ensuring database compatibility requires considering the specific language features and compatibility of different database systems. To tackle these challenges, we employ query rewriting techniques based on algebraic transformations, semantic analysis, splitting, and merging. This ensures query equivalence and enables the formulation of rules for equivalent query rewriting. Additionally, we ensure that the rewritten queries can be published under differential privacy scenarios, adhering to the constraints imposed by differential privacy. Furthermore, we guarantee database compatibility by formally expressing all queries with a unified structure and syntax, avoiding query syntax issues associated with different database types during the query equivalence rewriting process.

[*]Peng Tang is the corresponding author.

In terms of the novelty of our query rewriting, we explore this from two perspectives. Firstly, compared to traditional query rewriting methods that do not account for differential privacy, our approach differs in rewriting objectives, query types, and rewriting requirements. Traditional query rewriting rules focus on optimizing query execution efficiency, whereas our solution, within the context of differential privacy protection, aims to reduce the number of generated views to minimize the consumption of the privacy budget. Traditional methods generally address correlated nested queries, while our work not only includes correlated nested queries but also considers derived table queries and non-correlated nested queries, thus broadening the range of applicable query types. Traditional query rewriting primarily focuses on equivalence and compatibility, whereas our method, in addition to these basic requirements, emphasizes minimizing the number of generated views while ensuring differential privacy, thereby further optimizing the consumption of the privacy budget. Secondly, compared to existing query methods under differential privacy, the innovation in our approach lies in generating fewer views through query rewriting, which effectively reduces the consumption of the privacy budget and results in more accurate query outcomes. This method not only improves the efficiency of privacy budget usage but also enhances the practicality of the queries and the accuracy of the results. It is worth mentioning that some existing studies [15, 21, 28, 34] enhance the precision of query results by optimizing privacy budget allocation strategies. These techniques have provided valuable insights for our research. In future work, we plan to explore combining these techniques with our approach to achieve even better results.

Our key contributions are summarized as follows:

- We present a view generation approach based on query rewriting for multi-query operations across multiple relations while preserving differential privacy. This approach ensures that the number of views does not increase with changes in conditions within subqueries.
- We systematically classify derived table queries and nested queries and establish a comprehensive set of rewriting rules for them, guaranteeing query equivalence, differential privacy constraints, and database compatibility.
- We conduct an extensive experimental study over several datasets, demonstrating the practicality and desirable data utility of our proposed solution.

2 RELATED WORK

Differential privacy has gained wide application in database privacy queries, leading to significant achievements. To address the problem of a single query in single-relational databases, Geng et al. [12] created an optimal ϵ-differential privacy mechanism for single real-value queries. Liu et al. [23] introduced a mechanism based on generalized Gaussian distributions to optimize traditional differential privacy methods. Muthukrishnan et al. [26] proposed a differential privacy mechanism that improves privacy protection when publishing data by combining the properties of the Laplace and Gaussian distributions. These studies address the problem of noise addition in single-relational databases under differential privacy. However, while global sensitivity can be constrained in single-relational databases, it becomes challenging in multi-relational databases due to the presence of foreign key constraints.

To address the challenge of handling multi-query in private single-relational databases, Li et al. [22] introduced the Matrix Mechanism, an algorithm designed to optimize responses to linear count queries while maintaining differential privacy. Qiu et al. [30] proposed a new mechanism based on differential privacy to solve the problem of balancing the protection of individual privacy with the preservation of data utility when answering numerical queries. Pujol et al. [29] proposed a technique to address the issue of differential privacy in online query response in a multi-analyst environment. Kostopoulou et al. [19] developed the Turbo query engine, which optimizes the selection of noise parameters by using the improved private multiplication weights. These studies effectively addressed the problem of generating statistical results based on single relations under differential privacy, providing valuable insights for our work. However, the statistical results generated by a single relation are insufficient to represent the outcomes of complex queries involving joins, nested queries, and derived table queries in multi-relational databases.

To address the challenge of handling a single query in private multi-relational databases, Johnson et al. [17] presented FLEX, an aggregation query engine based on elastic sensitivity under differential privacy. Fang et al. [10] designed a universal mechanism, Shifted Inverse, which aims to provide a general differential privacy protection mechanism for any monotonic function. Dong et al. [3] proposed R2T, an aggregation query engine under differential privacy specifically designed to solve the challenging task of selecting truncation thresholds. Dong et al. [4] developed a novel method within the differential privacy framework using privacy composition techniques specifically designed to handle skewed and large-scale data. These studies effectively tackled the challenge of excessive global sensitivity in multi-relation queries under differential privacy, imparting valuable inspiration to our work. However, directly applying these methods to solve multi-query problems would suffer from the low query result accuracy caused by excessive privacy cost, and the inconsistent query results due to the impact of noise.

To tackle multi-query problems in private multi-relational databases, Kotsogiannis et al. [20] introduced PrivateSQL. To ensure a fixed privacy loss across all queries, PrivateSQL generates private synopses from multiple views over the base relation. However, PrivateSQL faces difficulties in effectively managing nested queries and derived table queries. Within these queries, subqueries may include filtering criteria. As these filtering criteria change, the views are consequently modified, resulting in an excessive proliferation of views. This, in turn, undermines the effectiveness of privacy cost reduction for PrivateSQL.

To achieve a more rational allocation of the privacy budget, Peng et al. [28] proposed Pioneer, which optimizes the use of the privacy budget by integrating historical query results with current data to create and select the most budget-efficient execution plans. Zhang et al. [34] developed DProvDB, which enhances query accuracy and ensures fair management of privacy loss through fine-grained privacy provenance and budget optimization. He et al. [15] proposed a fine-grained privacy provenance framework that tracks analysts' privacy loss and optimizes budget allocation to maximize accurate queries, ensuring fair and efficient privacy management in multi-analyst settings. Küchler et al. [21] introduced Cohere, which uses a unified interface and fine-grained management to efficiently allocate privacy budgets in complex systems, optimizing query performance while ensuring differential privacy. These studies offer multi-dimensional solutions for privacy budget management, advancing the integration of privacy protection and data analysis. In contrast, we reduce budget consumption and improve query accuracy by rewriting queries to minimize view generation. Future work will explore integrating these techniques to further optimize outcomes.

3 PRELIMINARIES

3.1 Database Queries

Let $\mathbf{R}$ denote a database schema comprising n relations $R_1, R_2, \ldots, R_n$, and p_i represent the attribute set in R_i. The multidimensional join relation, denoted as J, is defined as the join of these relations with their respective attribute sets:

$$J := R_1(p_1) \bowtie \cdots \bowtie R_n(p_n).$$

Let $\mathbf{D}$ be a database instance on schema $\mathbf{R}$, and $\mathbb{D}$ denote the set of all database instances under the schema $\mathbf{R}$. For each relation R in $\mathbf{R}$, $\mathbf{D}(R)$ represents the corresponding instance in $\mathbf{D}$, which is a physical relational instance of R. Let t be a tuple in $\mathbf{D}(R)$. For an aggregate query, the result can be expressed as: $Q(\mathbf{D}) = \sum_{q \in J(\mathbf{D})} \psi(q)$, where the function ψ depends on the query aggregation category. For example, for count queries, the function $\psi(\cdot)$ is equivalent to 1, and for aggregate queries such as SUM $(A * B)$, the function ψ equals $A * B$. This formula accommodates aggregate queries with arbitrary predicates by assigning $\psi(q) = 0$ if q does not satisfy the predicate.

3.2 Distributive Law

The distributive law [16] is widely used in logical expressions and is used to rewrite *AND* and *OR* operations to simplify or transform expressions. The distributive law can greatly simplify the form of logical operations, making the evaluation and understanding of expressions more intuitive. Specifically, the distributive law includes two forms: the distributive law of *AND* over *OR* and the distributive law of *OR* over *AND*:

$$A\ AND\ (B\ OR\ C) = (A\ AND\ B)\ OR\ (A\ AND\ C),$$

$$A\ OR\ (B\ AND\ C) = (A\ OR\ B)\ AND\ (A\ OR\ C).$$

By using these two forms, complex logical expressions can be effectively decomposed and simplified. The distributive law, as a fundamental rule in logical operations, has wide applications and significant importance.

3.3 Principle of inclusion-exclusion

The principle of inclusion-exclusion [13] is used to compute the size of the union of several sets, avoiding double counting of overlapping parts. For arbitrary finite sets $A_1, A_2, \ldots, A_n$, the general form of the inclusion-exclusion principle can be expressed as

$$\begin{aligned}\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| \\ + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| - \cdots \\ + (-1)^{n-1} |A_1 \cap \cdots \cap A_n|,\end{aligned}$$

where $|A|$ denotes the cardinality of the set A. For example, in the case of two sets, we can obtain the cardinality of their union by adding the cardinality of $|A|$ and $|B|$, and then subtracting the cardinality of their intersection. Using this general formula, we can accurately compute the size of the union of multiple sets without double-counting the intersecting parts.

3.4 Sensitivity in Differential Privacy

Differential privacy is a method that protects privacy by adding noise to query results For a function or query, its sensitivity [8] is defined as the maximum change in the function's output when an individual is added to or removed from the dataset. In other words, sensitivity measures how much the inclusion or exclusion of a single individual in the dataset affects the query result. A lower sensitivity indicates a lower risk of privacy loss for individuals in the dataset. Let F be a function that maps a data set to a real-valued vector of fixed length. For all neighboring data sets $\mathbf{D}$ and $\mathbf{D}'$, the sensitivity of the function F can be defined as $S(F) = \max_{\mathbf{D},\mathbf{D}'} \|F(\mathbf{D}) - F(\mathbf{D}')\|_1$, where $\|\cdot\|_1$ denotes the L_1 norm.

3.5 Laplace Mechanism in Differential Privacy

The Laplace mechanism [8] is a widely used technique in the field of differential privacy. The basic idea of the Laplace mechanism is to obfuscate query results by introducing noise from the Laplace distribution. Given a query function F with an output range of real numbers $\mathbb{R}$, the Laplace mechanism ensures that the function satisfies ϵ-differential privacy by adding noise to $F(\mathbf{D})$, resulting in

$$\tilde{F}(\mathbf{D}) = F(\mathbf{D}) + \text{Lap}\left(\frac{S(F)}{\epsilon}\right),$$

where $\tilde{F}(\mathbf{D})$ is the noisy query result, $\text{Lap}\left(\frac{S(F)}{\epsilon}\right)$ denotes a Laplace distribution with a mean of 0 and a scale parameter of $\frac{S(F)}{\epsilon}$, and $S(F)$ is the sensitivity of the query function F, which represents the maximum change in the function's output due to a single record change in the database.

The Laplace distribution is a double exponential distribution with a probability density function given by

$$\text{Lap}(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right),$$

where μ is the location parameter (usually set to 0) and b is the scale parameter. In the Laplace mechanism, the scale parameter b of the noise is inversely controlled by the privacy budget ϵ.

3.6 Composition Properties in Differential Privacy

To ensure data privacy in multiple queries and various application scenarios, the computation and management of privacy loss become a key issue in the study of differential privacy. The sequential composition property and the parallel composition property [8] of differential privacy are two important theoretical foundations that describe how to compute the total privacy loss under different query conditions.

The sequential composition property states that if multiple differential privacy mechanisms are applied sequentially to the same dataset, the total privacy loss is the sum of the privacy losses of each mechanism. Specifically, if there are k differential privacy mechanisms $M_1, M_2, \ldots, M_k$, each of which satisfies ϵ_i-differential privacy, then applying these mechanisms sequentially to the same dataset will result in an overall mechanism M that satisfies $\sum_{i=1}^{k} \epsilon_i$ differential privacy.

The parallel composition property, on the other hand, states that when multiple differential privacy mechanisms are applied to disjoint subsets of a dataset, the total privacy loss is determined by the maximum privacy loss among these mechanisms. Specifically, if there are k differential privacy mechanisms $M_1, M_2, \ldots, M_k$, each applied to disjoint subsets of the data and each satisfying ϵ_i differential privacy, then the parallel composition of these mechanisms M satisfies $max(\epsilon_1, \epsilon_2, \ldots, \epsilon_k)$ differential privacy. This is particularly useful for distributed computing or multi-party data sharing.

3.7 Differential Privacy in Multi-Relation Multi-Query

We employ differential privacy to protect personal privacy within a private multi-relational database, considering the constraint relations of foreign keys to define neighboring database instances. We designate the relation R_P as the primary privacy relation that requires protection, while any relation that directly or indirectly references R_P is considered a secondary privacy relation. Let $\mathbf{D}'$ be a neighboring database instance of $\mathbf{D}$, and $\mathbb{D}$ denote the set of all possible instances of the database under the R mode. In a private multi-relational database, $\mathbf{D}$ and $\mathbf{D}'$ are considered neighboring if and only if they differ only by a set of tuples that refer to the same $t_P \in \mathbf{D}(R_P)$. This relationship is denoted as $\mathbf{D} \sim \mathbf{D}'$.

For a random algorithm M, P_m represents the set of all possible values that algorithm M can produce. For any pair of neighboring datasets $\mathbf{D}$ and $\mathbf{D}'$, and any subset S_m of P_m, if the algorithm M satisfies the condition

$$\Pr\left[M(\mathbf{D}) \in S_m\right] \le e^{\epsilon} \Pr\left[M\left(\mathbf{D}'\right) \in S_m\right],$$

the algorithm M is said to satisfy ε-differential privacy, where ε is the privacy budget. A fundamental concept for achieving differential privacy is *sensitivity*. The global sensitivity GS_Q can be defined as the maximum change of a query considering any database instance. The global sensitivity GS_Q is then defined as follows:

$$GS_Q = \max_{\mathbf{D} \in \mathbb{D}, \mathbf{D}' \in \mathbb{D}, \mathbf{D} \sim \mathbf{D}'} \left|Q(\mathbf{D}) - Q\left(\mathbf{D}'\right)\right|.$$

The local sensitivity LS_Q defines the maximum change of a query considering a particular database instance. The local sensitivity LS_Q is then defined as follows:

$$LS_Q = \max_{\mathbf{D}' \in \mathbb{D}, \mathbf{D} \sim \mathbf{D}'} \left|Q(\mathbf{D}) - Q\left(\mathbf{D}'\right)\right|.$$

The downward local sensitivity defines the maximum change amount of the query by considering a particular database instance and requiring that neighboring database instances be contained in the particular database instance. The downward local sensitivity DLS_Q is then defined as follows:

$$DLS_Q = \max_{\mathbf{D}' \subseteq \mathbf{D}, \mathbf{D} \sim \mathbf{D}'} \left|Q(\mathbf{D}) - Q\left(\mathbf{D}'\right)\right|.$$

For a function F whose outputs are real, differential privacy can be achieved by the Laplace mechanism [8]. This mechanism works by adding Laplace noise to the true outputs. This paper addresses multi-query scenarios. In the case of a set of queries represented as $\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_k\}$, ensuring differential privacy requires dividing the privacy budget ε into $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k$, and adding Laplace noise with a scale $\lambda_i = S\left(Q_i\right)/\varepsilon_i$ to the true result for each query Q_i. By utilizing the composition properties [8] of differential privacy, the set of queries collectively maintains privacy guarantees. However, it should be noted that there is an inverse relationship between the privacy budget and the noise scale. As the privacy budget decreases, a larger noise scale may be necessary to achieve the same level of privacy protection, which in turn reduces the accuracy of the query results. Therefore, our primary focus is to effectively mitigate the consequences of privacy budget division.

4 VIEW GENERATION FOR MULTI-RELATION MULTI-QUERY WITH DIFFERENTIAL PRIVACY

To ensure a fixed privacy loss across all queries, a reasonable approach is to generate private synopses from multiple views over the base relations [20]. These synopses are tuned to have low error rates on a representative query workload. The system achieves this by answering queries on the private synopses, thereby minimizing additional privacy loss. However, the existing view generation-based approach faces difficulties in effectively managing correlated nested queries, non-correlated nested queries, and derived table queries. Within these queries, subqueries may include filter conditions. As these filter conditions change, the views are consequently modified, resulting in an excessive proliferation of views. This, in turn, accelerates the consumption of privacy budgets, undermining the effectiveness of privacy cost reduction. In particular, for correlated nested queries and non-correlated nested queries, the presence of subqueries leads to the problem of linear growth in the number of views. For derived table queries, the same problem of linear growth in the number of views can occur if there are filter conditions in the subquery.

5 QUERY REWRITING BASED VIEW GENERATION

To address the above issue, we propose a view generation approach based on **query rewriting**. The main idea of our solution is to transform the filter conditions of the subqueries into the filter conditions of the main query through query rewriting. And then, they can be eliminated without affecting the attributes that the view must contain. This will avoid a linear increase in the number of views caused by the changes in the subqueries filter conditions and decrease the privacy budget consumption. Note that in previous multi-relation queries, the filter conditions were placed in the main query. To reduce the number of views, we have removed the filter conditions. However, the current filter conditions are in subqueries. If we remove the filter conditions directly as before, the multi-relation query will degrade to a single-relation query, which is not desirable. For example, the following query, originally a multi-relation query, will become a single-relation query "SELECT Agg FROM customer" after removing the filter conditions of the subqueries. Then, the attribute *o_custkey* will no longer be contained in the generated view.

```
SELECT Agg FROM customer
WHERE 8 >
(SELECT Agg FROM orders WHERE o_custkey=c_custkey);
```

The concept of equivalent rewriting of correlated nested queries has been a popular research direction. However, previous work [5, 11, 18, 27, 33] has primarily focused on optimizing query efficiency. In contrast, we have different goals. We need to consider how to reduce the number of generated views under differential privacy conditions to improve query result accuracy. Additionally, there is limited research on equivalent query rewriting for other types of queries, such as non-correlated nested queries and derived table queries. Thus, we need to design novel query rewriting methods. Throughout this process, we need to consider the equivalence of queries before and after rewriting, compatibility across different database platforms, and differential privacy constraints.

To accomplish this, we first classify the queries into three categories: correlated nested queries, non-correlated nested queries, and derived table queries, based on the position and dependencies of the subqueries. We then further categorize these three types of queries according to predicates and other features, and we formalize all queries to ensure structural and syntactical consistency. This approach effectively prevents compatibility issues that might arise due to syntax differences across various database types during the process of equivalent query rewriting. Next, we conduct a comprehensive analysis of the logic, semantics, and potential edge cases of the queries, applying techniques such as algebraic transformations, semantic analysis, and query splitting and merging for query rewriting. Additionally, to facilitate understanding, we present these rewriting rules in increasing order of complexity. For instance, rewriting nested queries is typically more complex and often requires converting them into derived table form first, followed by moving the query conditions from the subquery to the main query. Therefore, we begin by introducing the rewriting rules for derived table queries and then delve deeper into the rewriting methods for nested queries.

Fig. 1 illustrates the query classification and basic process of query rewriting. For example, based on predicates, we can divide the correlated nested query into existence detection correlated nested query, set correlated nested query, in predicate correlated nested query, and comparison correlated nested query. Furthermore, we can divide set correlated nested queries based on set operations. Firstly, it is observed that alterations to the filter conditions of the subqueries incorporated in the main query cause a linear rise in the number of views. In order to tackle this concern, we proceed to rephrase the nested queries as derived table queries and chained queries. Subsequently, it is additionally discovered that modifications to the filter conditions within the subqueries similarly lead to a linear increase in the number of views. Consequently, we further refine the derived table queries, primarily by pushing down predicates and merging subqueries. In Fig. 1, the red number indicates the chapter in which we will introduce the query rewriting rules. For derived tables and nested queries, we first parse the query using sqlparse to generate an Abstract Syntax Tree (AST). Then, from the AST, we extract features and perform matching to accurately identify the query type. After that, through a series of predefined rewriting rules, we perform query equivalence rewritings while ensuring equivalence, compatibility, and compliance with differential privacy requirements.

In our solution, the rewriting process strictly adheres to the constraints of equivalence, compatibility, and differential privacy to ensure the validity and reliability of the final results. Firstly, regarding equivalence, we ensure that every transformation from the initial query to the final rewritten query is strictly equivalent, thereby maintaining overall semantic consistency. To this end, the paper provides detailed explanations for each rewriting rule to help readers fully understand the process and the rationale behind maintaining equivalence. Secondly, with respect to compatibility, we standardize and transform the rewriting process into algebraic expressions. By adopting this algebraic approach, we can maintain consistent transformation logic across different database systems, ensuring the scalability and portability of the solution. Finally, to meet differential privacy constraints, we avoid generating any standalone queries throughout the rewriting process, thereby preventing any potential impact on the differential privacy mechanism. At the same time, we rigorously validate the overall privacy guarantees of the approach in accordance with the formal definition of differential privacy.

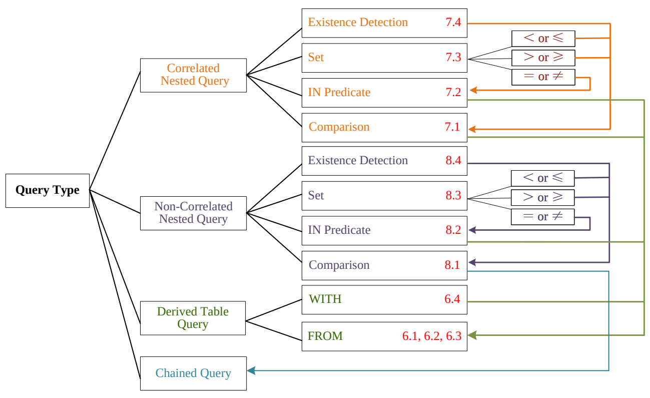


Figure 1: Query classification and basic process of query rewriting.

6 QUERY REWRITING FOR DERIVED TABLE QUERIES

A derived table query is a type of SQL query that includes a SELECT statement within another SELECT statement. The inner SELECT statement, also referred to as the subquery, is used to retrieve a set of results that are then used in the outer SELECT statement. The main goal of rewriting the derived table queries is to push down predicates, eliminate inner filter conditions containing OR types, and merge subqueries. In the following sections, we will explain how to handle various types of derived table queries. Firstly, we will explain how to rewrite FROM derived table queries in general. Next, we will explore methods for handling cases where multiple subqueries and OR-type filter conditions are present in FROM derived table queries. Finally, we will discuss how to rewrite WITH derived table queries.

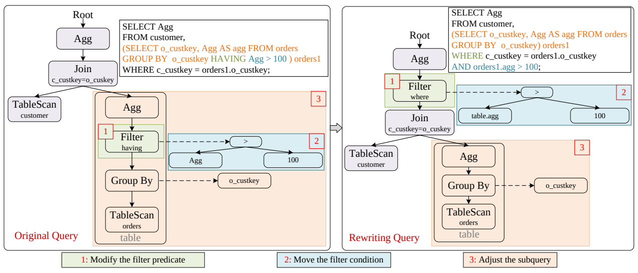
6.1 FROM Derived Table Query Rewriting

"From" derived table queries involve placing a subquery within the "from" clause. In these queries, we use the symbol $\otimes$ for a generic join operation, D to represent a relation in the subquery of a derived table, W for a "where" filter condition, and H for a "having" filter condition. Additionally, we have G_f^g for a group aggregation with a "group by" clause, where g represents the grouped column and f denotes the aggregation function. Similarly, G_f^1 represents a group aggregation without a "group by" clause. The symbol σ_ϕ denotes a filter with ϕ as the filter condition, and $\phi.at$ denotes the attribute in the comparison condition. Finally, T represents either a combination of multiple relations or a single relation.

$$T \otimes \left(\sigma_\phi G_f^g D\right) = \sigma_\phi \left(T \otimes G_f^g D\right), \text{ if } \left\{G_f^g = \emptyset\right\} \quad (1)$$

$$T\otimes\left(W_\phi G_f^g D\right) = W_\phi \left(T \otimes G_f^g D\right), \text{ if } \left\{\left(G_f^g \neq \emptyset\right) \& (\phi \cdot at = g)\right\} \quad (2)$$

$$T \otimes (H_\phi G_f^g \mathbf{D}) = (H \Rightarrow W)_\phi (T \otimes G_f^g D), \text{if } \{G_f^g \neq \emptyset\} \quad (3)$$



```
SELECT Agg
FROM customer,
(SELECT o_custkey, Agg AS agg FROM orders
GROUP BY  o_custkey HAVING Agg > 100 ) orders1
WHERE c_custkey = orders1.o_custkey;
```

```
SELECT Agg
FROM customer,
(SELECT o_custkey, Agg AS agg FROM orders
GROUP BY  o_custkey) orders1
WHERE c_custkey = orders1.o_custkey
AND orders1.agg > 100;
```

Figure 2: Example of FROM derived table query rewriting.

Now, let's go over the rules for handling "from" derived table queries:

- Rule (1) deals with derived table subqueries that don't have a "group by" predicate. The rule states that if there is no grouping involved in the subquery, we can move the filter conditions from the subquery to the main query. Since the filter and join operations are commutative, i.e., the filter σ_{ϕ} in $T \otimes D$ can be performed either before or after the join, Rule (1) is equivalent.
- Rule (2) explains how to handle derived table queries with both a "group by" predicate and a "where" filter condition in the subquery. In this case, we can move the filter conditions to the main query. Since the filter condition ϕ is on the grouping column g, applying the filter condition after the grouping is equivalent to applying it after the join. Thus, Rule (2) is equivalent. If both "where" and "having" filter conditions exist in the subquery, we apply Rule (1) first, followed by Rule (2).
- Rule (3) addresses derived table queries with a "group by" predicate and a "having" filter condition in the subquery. If the subquery involves grouping and the filter condition is of type "having," we can move the "having" filter condition to the "where" filter condition in the main query. Since the filter condition is applied to the aggregated results after grouping, applying it to the entire result set after grouping is equivalent to applying it after the join. That is, applying the "having" filter condition after grouping is equivalent to converting the "having" filter condition to a "where" filter condition and applying it after the join. Thus, Rule (3) is equivalent. Fig. 2 shows an example of rewriting a FROM derived table query using Rule (3).

6.2 FROM Derived Table Query Rewriting with Multiple Subqueries

Handling "from" derived table queries with multiple subqueries can lead to the generation of numerous temporary derived tables, resulting in high time and space complexity. To address this, we merge subqueries to reduce complexity.

The solution involves applying the rewrite rules to multiple subqueries within the "from" derived table query. There are two types of "from" derived table queries based on the presence or absence of a GROUP BY clause in the subquery.

Rule (4) handles "from" derived table queries with a GROUP BY clause. If multiple derived table subqueries have the same structure but different aggregation functions, they can be merged. According to the properties of grouping and aggregation in relational algebra, if multiple subqueries have the same grouping columns, we can combine multiple aggregate functions into a single grouping and aggregation operation. This means that performing multiple grouping and aggregation operations separately on the relation D is equivalent to performing a single grouping and aggregation operation with multiple aggregate functions. Hence, Rule (4) is equivalent.

$$\begin{aligned} &T \otimes G^{g}_{f_1} D \otimes G^{g}_{f_2} D = T \otimes G^{g}_{set(f_1,f_2)} D, \\ &\text{if } \left\{ \left(G^{g}_{f_1} \neq \emptyset \right) \& \left(G^{g}_{f_2} \neq \emptyset \right) \right\} \end{aligned} \tag{4}$$

Rule (5) handles "from" derived table queries without a GROUP BY clause. If multiple derived table subqueries have the same structure but different projection attributes, they can be merged. According to the properties of projection in relational algebra, if multiple subqueries have the same projection operation, we can combine multiple projection attributes into a single projection operation. This means that performing multiple projections separately on the relation D is equivalent to performing a single projection with multiple sets of projection attributes. Thus, Rule (5) is equivalent.

$$T \otimes G^{g}_{f} \pi_1 D \otimes G^{g}_{f} \pi_2 D = T \otimes G^{g}_{f} set(\pi_1, \pi_2) D, \text{ if } \left\{ G^{g}_{f} = \emptyset \right\} \tag{5}$$

For complex "from" derived table queries with both types of subqueries, we apply Rules (4) and (5) recursively until all subqueries are merged.

6.3 FROM Derived Table Query Rewriting Including OR-Type Filter Conditions

After rewriting the "from" derived table query, we may encounter OR-type filter conditions in the main query. Within the constraints of the differential privacy framework, for queries containing only AND-type filter conditions, we can directly compute statistical results to answer the user's query. However, when the filter conditions include OR-type queries, the direct computation of statistical results presents some difficulties.

Our main idea is to first rewrite the query using the distributive law so that all inner filter conditions use *AND*, while the outer filter conditions use *OR*. Specifically, we expand the *OR* conditions in the original query by converting them into multiple subqueries that contain only *AND* conditions, and then combine these subqueries using *OR*. According to the distributive law in logical algebra, these two filter conditions are equivalent, and therefore the filtering results applied to the relation D are also equivalent. Thus, Rule (6) is equivalent.

$$W_{(\phi_1 OR \phi_2) AND \phi_3} D = W_{(\phi_1 AND \phi_3) OR (\phi_2 AND \phi_3)} D \tag{6}$$

In query rewriting, we need to use the principle of inclusion-exclusion to split a query into multiple queries. The principle of inclusion-exclusion is a method often used in counting problems to help us accurately calculate the number of elements that satisfy certain conditions. According to the principle of inclusion-exclusion, to calculate the number of elements that satisfy multiple conditions, we can calculate the number of elements that satisfy each condition separately, then subtract the number of elements that satisfy two conditions simultaneously, and so on. According to the principle of inclusion-exclusion, the filtering results of the two expressions applied to the relation D are equivalent. Thus, Rule (7) is equivalent.

$$W_{\phi_1 OR \phi_2} D = W_{\phi_1} D + W_{\phi_2} D - W_{\phi_1 AND \phi_2} D \tag{7}$$

Using this method, we can convert a complex *OR* query into several simple *AND* queries, and then use the principle of inclusion-exclusion to combine the statistical results of these queries.

6.4 WITH Derived Table Query Rewriting

WITH derived table queries refer to queries where the subquery exists within the WITH clause. Let $W(D)$ represent the presence of a temporarily generated derived table within the WITH clause. Rule (8) describes the handling method for derived table queries. The main idea of Rule (8) is that if a derived table subquery exists in the WITH clause, it can be transferred to the FROM clause. Whether using the WITH clause or performing a subquery directly in the FROM clause, the result is to select tuples from the relationship D that satisfy the condition W and join them with T. The derived tables produced by both methods are logically equivalent, so their join operations should also be equivalent. Thus, Rule (8) is equivalent.

$$T \otimes \mathbb{W}(D) = T \otimes D \tag{8}$$

7 QUERY REWRITING FOR CORRELATED NESTED QUERIES

Correlated nested queries are a type of SQL query where the inner subquery references a column from the outer query. The correlation between the inner and outer queries is established through the use of a correlation variable or a shared column. The main goal of rewriting correlated nested queries is to remove the correlation and eliminate subqueries in the WHERE clause. This approach addresses the issue of linear growth in the number of views, resulting in improved query efficiency. In the following sections, we will explain how to handle various types of correlated nested queries.

7.1 Comparison Correlated Nested Query Rewriting

Comparison correlated nested queries involve filter conditions in the main query that include correlated subqueries and comparison operators. To express correlated nested queries using logical algebraic expressions, we introduce the $\mathbb{A}_{x_2}^{x_1}$ operator. This operator has two parameters: x_1 represents the join relation between the subquery and the main query, and x_2 represents the filter condition involving the subquery in the main query.

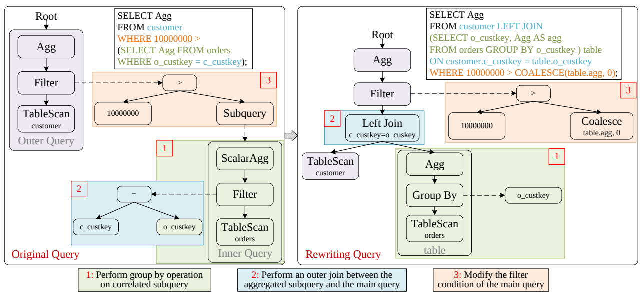
Let $\bowtie$ represent a natural join, and $\ltimes$ represent a left outer join. Let c denote a comparison filter condition or a comparison join condition, $c \cdot at$ denote the attribute in the comparison condition, $c \cdot sq$ denote the subquery in the comparison condition, and co denote a comparison condition of type "coalesce".

Rule (9) outlines the approach for handling comparison correlated nested queries. The rule involves performing an aggregation operation on the subquery, followed by a join between the aggregated subquery and the main query, and finally adding filter conditions to the main query.

Since there are no occurrences of null value tuples and the order of aggregation and join operations remains unchanged, the comparison condition c_2 is ultimately applied to the same intermediate result. Thus, Rule (9) is equivalent.

$$T_1 \mathbb{A}_{c_2}^{c_1} \left(G_f^1 T_2 \right) = \sigma_{c_2} \left(T_1 \bowtie_{c_1} \left(G_f^{c_2 \cdot at} T_2 \right) \right) \tag{9}$$

When using Rule (9) to handle comparison correlated nested queries, we also need to consider the issue of rewrite traps that may exist. Rewrite trap refers to a situation where the set of relational join keys involved in the main query is larger than the subquery, and the filter condition involving the subquery in the main query contains 0. The root cause of the rewrite trap is the failure to properly handle the occurrence of null value tuples.



```
SELECT Agg
FROM customer
WHERE 10000000 >
(SELECT Agg FROM orders
WHERE o_custkey = c_custkey);
```

```
SELECT Agg
FROM customer LEFT JOIN
(SELECT o_custkey, Agg AS agg
FROM orders GROUP BY o_custkey ) table
ON customer.c_custkey = table.o_custkey
WHERE 10000000 > COALESCE(table.agg, 0);
```

Figure 3: Example of comparison correlated nested query rewriting.

Rule (10) describes how to handle correlated nested queries correctly, accounting for both queries with and without rewrite traps. The approach involves performing an aggregation operation on the subquery, followed by an outer join between the aggregated subquery and the main query, and finally adding the "coalesce" filter condition to the main query. Since the aggregation and join operations in the left expression of Rule (10) effectively handle null value tuples, and the right expression ensures correct handling even if the tuples are empty through the left outer join and "coalesce". Thus, Rule (10) is equivalent regardless of the presence of rewrite traps.

By applying Rule (10), we can rewrite the comparison correlated nested query as a derived table query. For example, Fig. 3 shows a comparison correlated nested query rewriting process.

$$T_1 \mathbb{A}_{c_2}^{c_1} \left(G_f^1 T_2 \right) = \sigma_{c_2 \Rightarrow co_2} \left(T_1 \ltimes_{c_1} \left(G_f^{c_2 \cdot at} T_2 \right) \right) \tag{10}$$

7.2 IN Predicate Correlated Nested Query Rewriting

An "in" predicate correlated nested query refers to filter conditions in the main query that involve correlated subqueries using "in" and "not in" predicates. Let i represent an "in" predicate filter condition or "in" predicate join condition. Let πT denote a projection operation on relation T. Let $(\pi T) \cdot at$ represent the set of attributes resulting from the projection operation on relation T. Let $T \cdot ky$ represent the primary key of relation T. Let $i \Rightarrow =$ represent the conversion of the "in" predicate to the equals operator.

For rewriting "in" predicate correlated nested queries, one approach based on distinct rewriting is to join the subquery with the main query, add the filter condition on the main query, and perform a distinct operation on the entire query. However, publishing non-duplicate statistics for a set of queries under differential privacy is a challenging problem with no current solution. Thus, the distinct operation should be avoided when rewriting queries.

Our solution is to address the non-duplicate statistics problem by performing group by operations on subqueries. Rule (11) describes how to handle "in" predicate correlated nested queries. The rule involves performing a group by operation on the subquery, joining the grouped subquery with the main query, and adding the filter condition on the main query. The purpose of the group by operation is to prevent duplicate values after the join operation, making the two queries are no longer equivalent.

Table 1: Operator conversion rules.

	=	<>	<	<=	>	>=
ANY	IN	–	<MAX	<=MAX	>MIN	>=MIN
ALL	–	NOT IN	<MIN	<=MIN	>MAX	>=MAX

Since the projection and grouping operations select the same specific columns, i_1 and = are logically equivalent filter conditions under the grouping operation. Thus, Rule (11) is equivalent.

$$T_1 \mathbb{A}_{i_1}^{c_1} (\pi T_2) = \sigma_{i_1 \Rightarrow =} \left(T_1 \bowtie_{c_1} \left(G_{T_2 \cdot ky, (\pi T_2) \cdot at}^{T_2 \cdot ky, (\pi T_2) \cdot at} T_2 \right) \right) \quad (11)$$

7.3 Set Correlated Nested Query Rewriting

A "set" correlated nested query refers to filter conditions in the main query involving correlated subqueries using "ANY", "SOME", and "ALL" operators. Let s represent a set filter condition or set join condition. Please refer to Table 1 for the specific conversion correspondence. Rule 12 outlines how to handle set correlated nested queries. The rule involves adding an aggregate function to the subquery and converting the set operators in the subquery to comparison operators or "in" predicate operators. The conversion rules for set operators to comparison operators or "in" predicate operators are provided in Table 1. In Rule (12), the left expression uses a projection operation to select certain columns from the relation T_2, creates a join with the relation T_1 based on the comparison condition c_1, and applies the SET predicate s_1 as a filter condition. The right expression uses a grouping operation to select certain columns from the relation T_2, creates a join with the relation T_1 based on the comparison condition c_1, and applies $(c_2 \mid i_1)$ as a filter condition. The condition $s_1 \Rightarrow (c_2 \mid i_1)$ ensures that only those tuples that satisfy the SET predicate s_1 are retained. Since the projection and grouping operations select the same specific columns, s_1 and $(c_2 \mid i_1)$ are logically equivalent filter conditions. Thus, Rule (12) is equivalent.

$$T_1 \mathbb{A}_{s_1}^{c_1} (\pi T_2) = T_1 \mathbb{A}_{s_1 \Rightarrow (c_2 | i_1)}^{c_1} \left(G_{(\pi T_2) \cdot at}^{1} T_2 \right) \quad (12)$$

7.4 Existence Detection Correlated Nested Query Rewriting

Existence detection correlated nested query means that the filter conditions that involve correlated subqueries in the main query are EXISTS and NOT EXISTS. Let e represent the existence detection filter condition or existence detection join condition. Among them, ex refers specifically to the EXISTS filter or join condition, and nex refers specifically to the NOT EXISTS filter or join condition. Let cn represent an aggregate function of type "count". Rules (13) and (14) together describe how to handle existence detection correlated nested queries. The main idea of Rules (13) and (14) is to first add an aggregation function to the subquery, and then convert the existence detection filter conditions into comparison filter conditions according to the semantics of the query. In Rule (13), the EXISTS semantics of the left expression checks whether there is at least one tuple that satisfies the condition. The right expression achieves the same effect by checking whether the count of grouped tuples is greater than or equal to 1. Thus, the expressions on the left and right sides of Rule (13) have the same semantics and are equivalent. In Rule (14), the NOT EXISTS semantics of the left expression checks that no tuples satisfy the condition. The right expression achieves the same effect by checking whether the count of grouped tuples is less than 1. Thus, the expressions on the left and right sides of Rule (14) have the same semantics and are equivalent.

$$T_1 \mathbb{A}_{ex_1}^{c_1} T_2 = T_1 \mathbb{A}_{ex_1 \Rightarrow c_2 \cdot sq \geq 1}^{c_1} \left(G_{cn}^{1} T_2 \right) \quad (13)$$

$$T_1 \mathbb{A}_{nex_1}^{c_1} T_2 = T_1 \mathbb{A}_{nex_1 \Rightarrow c_2 \cdot sq < 1}^{c_1} \left(G_{cn}^{1} T_2 \right) \quad (14)$$

8 QUERY REWRITING FOR NON-CORRELATED NESTED QUERIES

A non-correlated nested query is a type of SQL query where the subquery is not directly dependent on the outer query. In a non-correlated nested query, the subquery can be executed independently, generating a result set that is then used by the outer query. Correlated nested queries and non-correlated nested queries have distinct differences. In summary, correlated nested queries have a direct dependency on the outer query and are re-evaluated for each row, while non-correlated nested queries are independent of the outer query and are executed once, producing a result that is then used by the outer query. Thus, they have different rewriting rules. The main goal of rewriting non-correlated nested queries is to remove the subqueries from the WHERE clause, effectively addressing the issue of linear growth in the number of views. Below, we will introduce the rewriting rules for non-correlated nested queries. On one hand, this aims to demonstrate the approach for rewriting non-correlated nested queries. On the other hand, it facilitates understanding the differences between the rewriting rules for correlated nested queries.

8.1 Comparison Non-Correlated Nested Query Rewriting

A comparison non-correlated nested query refers to filter conditions in the main query involving non-correlated subqueries using comparison operators. To express non-correlated nested queries using logical algebraic expressions, we introduce the operator $\mathbb{B}_x$, where x represents the filter conditions involving subqueries in the main query. Let $\rightarrow$ denote the sequential execution of chained queries. Rule (15) outlines how to handle comparison non-correlated nested queries.

The main idea of Rule (15) is to assign the subquery to a variable, replace the subquery in the main query with the assigned variable, and then execute the subquery and the main query sequentially to obtain the query result. The query resulting from splitting and combining the comparison non-correlated nested query using Rule (15) is called a chained query. In cases where multiple nested relationships exist between the subqueries and the main query, the principle is to build the chained query from the inside out, with the result of the entire chained query being the result of the outermost query in the chain. For subsequent processing of chained queries, each query in the chain can be treated as an independent query. These independent queries can be processed in the same way as normal queries.

In the left and right expressions of Rule (15), the subquery $G_f^1 T_2$ represents the aggregation operation on the relation T_2, resulting in the same execution result. In the left expression, the relation T_1 uses the comparison condition c_1 to filter the subquery results. In the right expression, the relationship T_1 uses the comparison condition c_1 to filter the variable v. Because the variable v is assigned the result of the subquery, the filter condition in the left expression is equivalent to the filter condition in the right expression. In addition, the chained execution order in the right expression ensures the same execution order as in the left expression. Thus, Rule (15) is equivalent.

$$T_1 \mathbb{B} c_1 \left(G_f^1 T_2 \right) = \left(\left(v := G_f^1 T_2 \right) \rightarrow \left(\sigma_{c_1 \cdot sq := v} T_1 \right) \right) \tag{15}$$

8.2 IN Predicate Non-Correlated Nested Query Rewriting

An "in" predicate non-correlated nested query refers to filter conditions in the main query involving non-correlated subqueries using "in" and "not in" operators. Let *cndt* stand for "distinct" non-repeating count. To maintain the same semantics when rewriting "in" predicate non-correlated nested queries, we should add "distinct" parameters in the SELECT clause. However, under differential privacy, publishing non-duplicate statistics for a set of queries is currently unsolved. We discovered that if the ordinary count and distinct count of the subquery are equal, the ordinary and distinct aggregate results of the rewritten query are also equal.

Rule (16) describes how to handle "in" predicate non-correlated nested queries and the conditions for its application. The rule converts the non-correlated subquery into a derived table subquery, transforms the "in" predicate filter condition into an equal operator filter condition, and joins the derived table subquery with the main query. The rule applies when the ordinary count and distinct count of the subquery are equal. According to the premise of Rule (16), if the regular count of the subquery $G_{cn}^1 T_2$ and the distinct count $G_{cndt(\pi T_2 \cdot at)}^1 T_2$ are equal, it means that there are no duplicate values in T_2. Since there are no duplicate values in T_2, the equality predicate = and the "in" predicate i are equivalent in this case. Thus, Rule (16) is equivalent.

$$T_1 \mathbb{B}_i T_2 = T_1 \ltimes_{i \Rightarrow =} T_2, \text{if} \left\{ \left(G_{cn}^1 T_2 \right) = \left(G_{cndt(\pi T_2 \cdot at)}^1 T_2 \right) \right\} \tag{16}$$

Rule (17) presents an alternative approach to handling "in" predicate non-correlated nested queries, along with its limiting conditions. The rule groups the non-correlated subquery, converts the "in" predicate filter condition into an equal operator filter condition, and joins the grouped subquery with the main query. The rule applies when there are no filter conditions in the non-correlated subquery or when the projection attributes of the subquery are the same as the filter attributes. According to the premise of Rule (17), if there are no filter conditions in the subquery, or if the attributes of the filter conditions are the same as the projection attributes, this means that the output of the subquery is determined entirely by its projection attributes. Since the attributes of the filter conditions and the projection attributes are consistent, the equality predicate and the "in" predicate are equivalent in this case. Thus, Rule (17) is equivalent.

$$\begin{aligned} &T_1 \mathbb{B}_i \left(\sigma_{c_1} T_2 \right) = T_1 \ltimes_{i \Rightarrow =} \left(\sigma_{c_1} G_{\pi T_2 \cdot at}^{\pi T_2 \cdot at} \right) T_2, \\ &\text{if} \left\{ \left(\sigma_{c_1} = \emptyset \right) \mid \left(c_1 \cdot at = \pi T_2 \cdot at \right) \right\} \end{aligned} \tag{17}$$

If there are no filter conditions in the non-correlated subquery or if the projection attributes of the subquery match the filter attributes, we use Rule (17) to rewrite the "in" predicate non-correlated nested query. If the ordinary count and distinct count of the subquery are equal, we use Rule (16) to rewrite the query.

8.3 Set Non-Correlated Nested Query Rewriting

Set non-correlated nested query means that the filter conditions that involve non-correlated subqueries in the main query are "ANY", "SOME", and "ALL". Rule (18) describes the method for handling set non-correlated nested queries. The main idea of Rule (18) is to first add aggregate functions for the non-correlated subquery, and then convert set operators involving subquery filter conditions to comparison operators or "in" predicate operators. For specific conversion correspondence rules, see Table 1. The main idea of the equivalence proof for Rule (18) is similar to that of Rule (12).

$$T_1 \mathbb{B}_{s1} \left(\pi T_2 \right) = T_1 \mathbb{B}_{s1 \Rightarrow (c_1 | i_1)} \left(G_{(\pi T_2) \cdot at}^1 T_2 \right) \tag{18}$$

8.4 Existence Detection Non-Correlated Nested Query Rewriting

Existence detection non-correlated nested query means that the filter conditions that involve non-correlated subqueries in the main query are EXISTS and NOT EXISTS. Rules (19) and (20) describe how to handle existence detection non-correlated nested queries. The main idea of Rules (19) and (20) is to first add aggregation functions for the non-correlated subquery, and then convert the existence detection filter conditions involving the non-correlated subquery into comparison filter conditions according to the semantics of the query. The main idea of the equivalence proof for Rules (19) and (20) is similar to that for Rules (13) and (14).

$$T_1 \mathbb{B}_{ex_1} T_2 = T_1 \mathbb{B}_{ex_1 \Rightarrow c_2 \cdot sq \geq 1} \left(G_{cn}^1 T_2 \right) \tag{19}$$

$$T_1 \mathbb{B}_{nex_1} T_2 = T_1 \mathbb{B}_{nex_1 \Rightarrow c_2 \cdot sq < 1} \left(G_{cn}^1 T_2 \right) \tag{20}$$

9 SYSTEM IMPLEMENTATION

As illustrated in Fig. 4, we design a differentially private SQL query engine, **ViewRewrite** [6], based on our query rewriting-based view generation module.

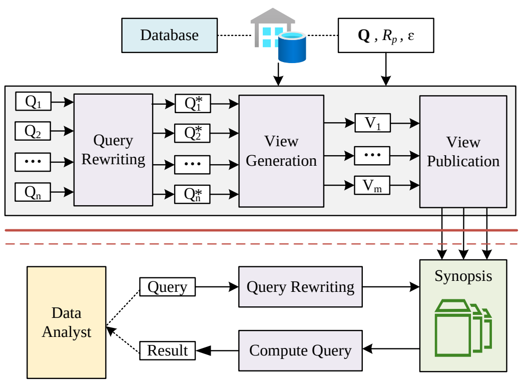


Figure 4: System structure.

The engine comprises three modules: **query rewriting**, **view generation**, and **view publication**. **Query rewriting** module rewrites SQL queries according to our specific rules to address the issue of linear growth in the number of views for correlated nested queries, non-correlated nested queries, and derived table queries. **View generation** module merges similar subqueries to further reduce the number of views generated and improve solution accuracy, and **view publication** refers to generating differentially private statistical results (i.e., synopsis) for each view. The process of view generation and view publication is similar to that described in PrivateSQL [20], with a few modifications.

In particular, in the view publication module, we need to first determine the truncation threshold of the view to determine the scale of noise injected into the synopsis. In this work, we determine the truncation threshold of the view based on the downward local sensitivity [3] and the sparse vector technique [8]. Firstly, for a database instance $\mathbf{D}$, calculate the downward local sensitivity DLS_Q of query Q. The calculation method for downward local sensitivity:

$$DLS_Q = \max_{t_p \in \mathbf{D}(R_p)} S_Q(\mathbf{D}, t_P),$$

$$S_Q(\mathbf{D}, t_P) := \sum_{q \in J(\mathbf{D})} \psi(q) \mathbb{I}(q \text{ references } t_P).$$

Here, $S_Q(\mathbf{D}, t_P)$ represents the sensitivity calculation of the tuple $t_p \in \mathbf{D}(R_p)$, $\mathbb{I}(q$ references$\cdot$ $t_P)$ denotes the query q referencing the tuple t_P, $J(\mathbf{D})$ denotes the multidimensional join relationship, $\psi(q)$ represents the type of aggregate query, $\mathbf{D}$ represents the database instance, and R_p represents the primary privacy relationship. Then, add Laplace noise [8] with sensitivity DLS_Q and privacy budget ϵ_1 to the original query yields $\hat{Q}(\mathbf{D}) = Q(\mathbf{D}) + \text{lap}(DLS_Q/\epsilon_1)$. Next, for each candidate truncation threshold τ, calculate $q_\tau = \frac{Q_\tau(\mathbf{D}) - \hat{Q}(\mathbf{D})}{\tau}$, where $Q_\tau(\mathbf{D})$ denotes the query result with τ. Finally, use the privacy budget ϵ_2 to find the first q_τ greater than 0 using sparse vector technology and return the τ corresponding to q_τ as the truncation threshold.

Privacy guarantee. It is important to highlight that our proposed solution ViewRewrite intelligently allocates the privacy budget among the generated views and subsequently generates differentially private synopses based on these views, ensuring end-to-end differential privacy guarantee. Then, we will provide a comprehensive proof of the privacy guarantee.

In particular, our solution ViewRewrite comprises four key stages: query rewriting, view generation, view publication, and query answering. Notably, only the view publication stage consumes the privacy budget. This stage involves two primary components: calculating view truncation thresholds and generating statistical synopses. By adhering to the principle of sequential composition, we effectively allocate the privacy budget within this stage. In the truncation threshold calculation phase, the sparse vector mechanism is employed to determine appropriate thresholds, thereby effectively enhancing the solution's accuracy. This mechanism incorporates the concept of downward local sensitivity from R2T [3], making the determination of truncation thresholds more precise. Since the sparse vector mechanism must satisfy differential privacy requirements, the computation process consumes a portion of the privacy budget to ensure strict adherence to differential privacy constraints throughout the calculation. In the synopsis generation phase, we apply the matrix mechanism to process the original statistical results under differential privacy. By introducing Laplace noise, the matrix mechanism effectively protects the privacy of data subjects while ensuring the utility of the statistical results. Within the sequential and parallel combination structures, the matrix mechanism rigorously adheres to the constraints of differential privacy, thereby ensuring that the final published statistical synopsis complies with differential privacy requirements.

Effectiveness analysis. Throughout the solution, the privacy budget is consumed when synopses are generated from views. Our solution transforms the filter conditions of the subqueries into the filter conditions of the main query through query rewriting. And then, the filter conditions can be eliminated without affecting the attributes that the view must contain. This will avoid a linear increase in the number of views caused by the changes in the subquery filter conditions and decrease the privacy budget consumption.

10 EXPERIMENTAL EVALUATION

We first evaluate our approach's performance across various workloads, and then compare it with PrivateSQL's performance on the workloads it supports, to ensure fairness.

10.1 Experimental Setup

Databases. We evaluate the performance of our approach ViewRewrite using two publicly available datasets, namely TPC-H [32] and U.S. Census [31]. TPC-H benchmark with a schema consisting of 8 relationships, including `Customer`, `Orders`, and `Lineitem`, etc. U.S. Census dataset with the following schema: `Household` and `Person`.

Privacy Policy. We allow data owners to choose appropriate privacy protection policies based on actual situations. For the TPC-H schema we have chosen `Customer`, `Orders`, and `Lineitem` as the primary privacy protection relationships for testing. For U.S. Census schema we have chosen `Household` as primary privacy protection relationships for testing.

Privacy Budget. The privacy budget in differential privacy is a parameter used to quantify the upper limit of privacy loss in data disclosure. We evaluated the solution with privacy budgets of 0.5, 1, 2, 4, 8, 16, 32.

Workload. We call a set of queries used for solution testing a workload. There are 31 workloads available for experimental testing. Among them, 11 are used for overall experimental analysis, 5 for comparative experiments, and 15 for ablation experiments. It is important to note that these 31 workloads are not derived from standard benchmark suites (such as TPC-H), as the queries in existing benchmarks do not meet the specific needs of our experiments. Therefore, we have specially designed these workloads to better support the objectives of our experiments. The following will provide a detailed description of these workloads. W1-W30 are used to test the TPC-H database. W1-W5, W6-W10, W11-W15 each contain 750, 1500, 3000, 6000, 12000 queries, respectively. W16-W20, W20-W25, W25-W30 each contain 200, 400, 800, 1600, 3200 queries, respectively. W1-W10 are used for overall analysis experiment. W1-W5 are of the count type, while W6-W10 are of the sum type. Each of W1-W10 includes single-relation queries, join queries, nested queries, and derived table queries. W11-W15 are used for comparative experiments and belong to the count type. Each of W11-W15 includes single-relation queries, join queries without self-join and non-equivalence join, comparison correlated nested queries without query rewriting traps, non-correlated nested queries, and derived table queries. W16-W30 are used for the ablation experiment. W16-W20 represent correlated queries, W21-W25 represent non-correlated queries, and W26-W30 represent derived table queries. W31 is used to test U.S. Census, containing 3000 queries that include single-relation queries, join queries, nested queries, and derived table queries.

Competitor. We compare our solution with PrivateSQL proposed by Kotsogiannis et al. [20], which is currently the state-of-the-art solution for answering multi-type aggregate queries based on differential privacy in multi-relational databases. To ensure fairness, we chose the W11-W15 workloads supported by PrivateSQL for the solution comparison. We obtain the source code of PrivateSQL via email from the authors and test it under the same workload as ours, rather than through simulation.
Default Settings. By default, with TPC-H as the database, the privacy budget is 8. For the TPC-H database, we set the size to 10M, and the privacy policy to `Orders`. The workload for the overall analysis experiment is W7, and the workload for the comparison experiment is W12. In the ablation experiment, the workload used to test the impact of query rewriting on correlated queries is W17, the workload used to test the impact of query rewriting on non-correlated queries is W22, and the workload used to test the impact of query rewriting on derived table queries is W27. For the U.S. Census database, we set the size to 10M, the privacy policy to `Household`, the workload to W31.
Metrics. We use relative error, number of views, synopsis generation time, and query response time to evaluate the experiments. For a query Q, let $y = Q(\mathbf{D})$ be its true answer, and $\tilde{y}$ be a noisy answer, we define the relative error of $\tilde{y}$, as $\text{Error}(y, \tilde{y}) = |y - \tilde{y}|/\max(50, y)$.

10.2 Overall Analysis

We conducted evaluations of our solution using various databases, sizes, privacy policies, and workloads. In Fig. 5a, we observe that the scheme's overall relative error decreases as the size of the database instance increases. This is due to the expanding range of query results with larger database instances, while the total amount of noise remains constant. Consequently, the impact of noise decreases, resulting in a median relative error of less than 0.01 when the database instance is 40M or greater. Fig. 5b illustrates the effect of different privacy policies on the overall relative error of the solution. The `Customer` privacy policy provides the highest level of privacy protection, while the `Lineitem` privacy strategy has the lowest error. Data publishers should select an appropriate privacy policy based on their specific circumstances. In Fig. 5c, we observe that increasing the privacy budget leads to a decrease in the overall relative error of the solution. When the privacy budget is 4 or greater, the median relative error of the solution is less than 0.1. Regarding workloads, Fig. 5d and 5e show that the relative error of the solution is largely unaffected by different count types and sum types of workloads. Regardless of the count or sum types, the solution consistently generates 15 and 14 views, respectively. Under all different workloads, the median relative error for count and sum types is significantly less than 0.1. Fig. 5f shows the performance of our approach on the U.S. Census. It is worth noting that our approach demonstrates similar performance on both datasets.

10.3 Comparison with PrivateSQL

We compared our solution with PrivateSQL across different database sizes, privacy policies, and workloads. To ensure fairness, we chose the W11-W15 workloads supported by PrivateSQL for the solution comparison.

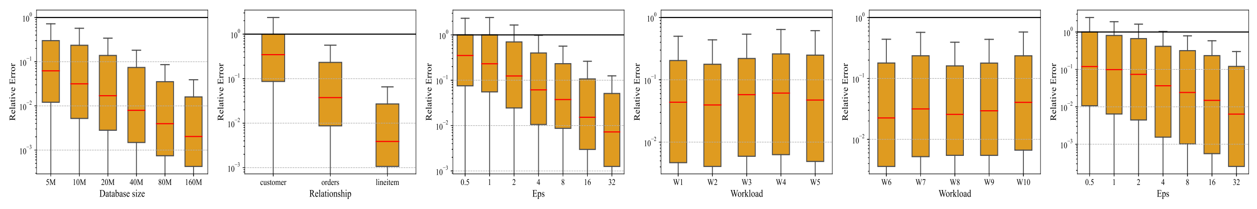
In Fig. 6a, we observe that both our solution and PrivateSQL exhibit a decreasing overall relative error as the number of database instances increases. However, our solution demonstrates a faster decrease in relative error compared to PrivateSQL. Across all different database instances, our solution consistently achieves significantly lower relative error than PrivateSQL. Fig. 6b demonstrates that our solution consistently achieves significantly lower relative error than PrivateSQL under different privacy policies. Fig. 6c reveals a trend where both our solution and PrivateSQL experience a decreasing overall relative error as the privacy budget increases. However, our solution consistently achieves significantly lower relative error than PrivateSQL across all different privacy budgets. Fig. 6d presents the overall relative error distribution of our solution and PrivateSQL under different workloads. As the number of queries in the workload increases, our solution maintains a nearly constant overall relative error, while PrivateSQL exhibits an increasing trend. This behavior arises because the number of views generated by our solution remains constant at 14, while PrivateSQL shows a rapid increase, as depicted in Fig.6e. Our solution consistently achieves significantly lower overall relative error than PrivateSQL under all different workloads. Fig. 6f showcases the synopsis generation time and total query response time of our solution and PrivateSQL for different workload sizes. Among them, the synopsis generation time consists of three parts: query rewriting time, view generation time, and view publication time. Our solution generates views with more tuples and attributes compared to PrivateSQL. Consequently, the query response time of our solution is slightly higher than that of PrivateSQL. However, the number of views generated by our solution is always much smaller than that of PrivateSQL, thereby reducing the synopsis generation time for our solution. With an increasing number of workload sizes, the advantage becomes increasingly pronounced. Across all different workload sizes, the total time of our solution is always less than that of PrivateSQL. This is because PrivateSQL typically generates a separate view for each query, rapidly proliferating views. In contrast, our solution controls view growth effectively through query rewriting. Moreover, our query rewriting involves only equivalence transformations, with negligible computational overhead compared to query execution.

10.4 Ablation Experiment

We evaluate our solution on nested queries and derived table queries respectively. We compare the performance of ViewRewrite and PrivateSQL in terms of relative error, number of views generated, and time, based on various database sizes, privacy policies, privacy budgets, and workloads. The performance of the two solutions on nested queries and derived table queries is broadly consistent with their performance on all query types. Detailed experimental data are shown in Table 2.

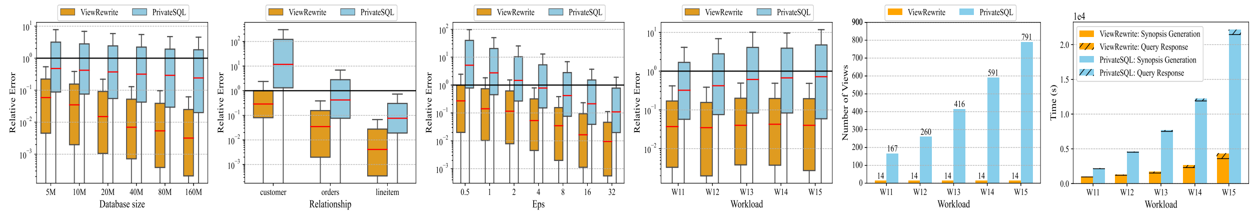
11 CONCLUSION

This research proposes a query rewriting-based view generation approach to efficiently handle multi-relation multi-query while preserving differential privacy. The approach effectively addresses the challenge of excessive view proliferation by performing equivalent rewriting on nested and derived table queries. Extensive experiments on real datasets demonstrate the desirable data utility and minimal computational costs of the proposed solution. The findings highlight the effectiveness of the approach in achieving privacy preservation while improving result accuracy and ensuring compatibility across different database platforms.



(a) Database size (b) Relationship (c) Eps (d) Count-type (e) Sum-type (f) U.S. Census

Figure 5: Relative error of ViewRewrite under different settings.



(a) Error, database size (b) Error, relationship (c) Error, eps (d) Error, workload (e) View, workload (f) Time, workload

Figure 6: Results of ViewRewrite and PrivateSQL under different settings.

Table 2: The impact of query rewriting on nested and derived table queries.

Evaluation metrics	Different setting		Correlated query		Non-correlated query		Derived query	
			ViewRewrite	PrivateSQL	**ViewRewrite**	PrivateSQL	**ViewRewrite**	PrivateSQL
Median relative error	Database size	10M	0.021218	0.986581	0.021009	1.135431	0.029882	3.035412
		20M	0.007973	0.696718	0.017655	1.035596	0.019987	2.513428
		40M	0.004352	0.448355	0.010665	0.835515	0.009236	2.055838
		80M	0.002707	0.248355	0.007910	0.664496	0.005788	1.578090
	Relationship	customer	0.181899	26.279078	0.296397	39.851007	0.334546	73.641692
		orders	0.021289	0.792546	0.041098	1.235792	0.056836	2.9527641
		lineitem	0.005289	0.091567	0.008754	0.124694	0.010296	0.2081002
	Eps	1	0.091912	6.111440	0.140001	8.29277	0.189277	15.307424
		4	0.039133	1.709608	0.086149	2.087193	0.059709	5.826856
		8	0.013172	0.883628	0.025544	1.106706	0.029117	2.913428
		16	0.007452	0.442263	0.012992	0.574895	0.016192	0.968714
	Workload size	400	0.021759	0.846582	0.024097	1.278901	0.049293	2.418623
		800	0.029236	1.057762	0.028842	2.030648	0.053509	4.450558
		1600	0.017365	1.243963	0.025430	2.552517	0.050332	5.759089
		3200	0.021732	1.626135	0.026163	3.678376	0.045816	7.771597
Number of views	Workload size	400	4	81	4	87	4	124
		800	4	119	4	135	4	185
		1600	4	160	4	182	4	273
		3200	4	194	4	231	4	357
Synopsis generation time (s)	Workload size	400	363.79	1378.09	372.07	1406.65	394.67	1479.98
		800	462.05	2299.72	472.30	2344.72	497.91	2465.95
		1600	696.10	3629.64	711.02	3701.24	747.57	3888.30
		3200	1086.67	6549.35	1109.40	6681.34	1168.87	7019.41
Query response time (s)	Workload size	400	29.50	24.16	30.89	25.63	35.05	28.86
		800	58.22	51.30	61.58	53.77	68.89	59.89
		1600	113.30	99.68	118.88	103.91	132.06	114.96
		3200	233.97	214.25	244.42	223.91	272.48	247.86
Total time (s)	Workload size	400	393.29	1402.25	402.96	1432.29	429.73	1508.85
		800	520.28	2351.03	533.88	2398.50	566.80	2525.84
		1600	809.40	3729.33	829.90	3805.15	879.64	4003.27
		3200	1320.65	6763.61	1353.83	6905.25	1441.36	7267.27

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments which significantly improved this paper. The work was supported by National Natural Science Foundation of China under Grant No. 62002203, No. 62072136, No. 62372268, Major Scientific and Technological Innovation Projects of Shandong Province, China No. 2024CXGC010114, Shandong Provincial Natural Science Foundation, China No. ZR2020QF045, No. ZR2022LZH013, No. ZR2021LZH007, Fundamental Research Funds for the Central Universities No. 3072020CFT2402, Young Scholars Program of Shandong University, Department of Science&Technology of Shandong Province grant No. SYS202201, and Quan Cheng Laboratory grant No. QCLZD202302.

REFERENCES

[1] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnés, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 441–459.

[2] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. 2017. Collecting Telemetry Data Privately. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 3571–3580.

[3] Wei Dong, Juanru Fang, Ke Yi, Yuchao Tao, and Ashwin Machanavajjhala. 2023. R2T: Instance-optimal Truncation for Differentially Private Query Evaluation with Foreign Keys. *SIGMOD Rec.* 52, 1 (2023), 115–123.

[4] Wei Dong, Dajun Sun, and Ke Yi. 2023. Better than Composition: How to Answer Multiple Relational Queries under Differential Privacy. *Proc. ACM Manag. Data* 1, 2 (2023), 123:1–123:26.

[5] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.

[6] Xinglin Du, Peng Tang, Rui Chen, Ning Wang, Chengyu Hu, and Shanqing Guo. 2024. ViewRewrite: A Differentially Private SQL Query Engine for Efficient Multi-Relation Multi-Query. https://github.com/xinglindu/ViewRewrite

[7] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 3876. Springer, 265–284.

[8] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3-4 (2014), 211–407.

[9] Cynthia Dwork, Guy N. Rothblum, and Salil P. Vadhan. 2010. Boosting and Differential Privacy. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 51–60.

[10] Juanru Fang, Wei Dong, and Ke Yi. 2022. Shifted Inverse: A General Mechanism for Monotonic Functions under User Differential Privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 1009–1022.

[11] Philipp Fent, Guido Moerkotte, and Thomas Neumann. 2023. Asymptotically Better Query Optimization Using Indexed Algebra. *Proc. VLDB Endow.* 16, 11 (2023), 3018–3030.

[12] Quan Geng and Pramod Viswanath. 2016. The Optimal Noise-Adding Mechanism in Differential Privacy. *IEEE Trans. Inf. Theory* 62, 2 (2016), 925–951.

[13] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley.

[14] Samuel Haney, Ashwin Machanavajjhala, John M. Abowd, Matthew Graham, Mark Kutzbach, and Lars Vilhuber. 2017. Utility Cost of Formal Privacy for Releasing National Employer-Employee Statistics. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1339–1354.

[15] Xi He and Shufan Zhang. 2024. Differential Privacy with Fine-Grained Provenance: Opportunities and Challenges. *IEEE Data Eng. Bull.* 47, 2 (2024), 21–49.

[16] Michael Huth and Mark Dermot Ryan. 2004. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press.

[17] Noah M. Johnson, Joseph P. Near, and Dawn Song. 2018. Towards Practical Differential Privacy for SQL Queries. *Proc. VLDB Endow.* 11, 5 (2018), 526–539.

[18] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2023. Correction to: Data dependencies for query optimization: a survey. *VLDB J.* 32, 2 (2023), 471.

[19] Kelly Kostopoulou, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. 2023. Turbo: Effective Caching in Differentially-Private Databases. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 579–594.

[20] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *Proc. VLDB Endow.* 12, 11 (2019), 1371–1384.

[21] Nicolas Küchler, Emanuel Opel, Hidde Lycklama, Alexander Viand, and Anwar Hithnawi. 2024. Cohere: Managing Differential Privacy in Large Scale Systems. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 991–1008.

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

[23] Fang Liu. 2019. Generalized Gaussian Mechanism for Differential Privacy. *IEEE Trans. Knowl. Data Eng.* 31, 4 (2019), 747–756.

[24] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. 2009. Computational Differential Privacy. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, Vol. 5677. Springer, 126–142.

[25] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. 2009. Computational Differential Privacy. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Vol. 5677. Springer, 126–142.

[26] Gokularam Muthukrishnan and Sheetal Kalyani. 2023. Grafting Laplace and Gaussian Distributions: A New Noise Mechanism for Differential Privacy. *IEEE Trans. Inf. Forensics Secur.* 18 (2023), 5359–5374.

[27] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. *Datenbanksysteme für Business, Technologie und Web (BTW)* P-241 (2015), 383–402.

[28] Shangfu Peng, Yin Yang, Zhenjie Zhang, Marianne Winslett, and Yong Yu. 2013. Query optimization for differentially private data management systems. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 1093–1104.

[29] David Pujol, Albert Sun, Brandon Fain, and Ashwin Machanavajjhala. 2022. Multi-Analyst Differential Privacy for Online Query Answering. *Proc. VLDB Endow.* 16, 4 (2022), 816–828.

[30] Yuan Qiu, Wei Dong, Ke Yi, Bin Wu, and Feifei Li. 2022. Releasing Private Data for Numerical Queries. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 1410–1419.

[31] William Sexton, John M. Abowd, Ian M. Schmutte, and Lars Vilhuber. 2017. Synthetic population housing and person records for the United States. https://doi.org/10.3886/E100274V1.

[32] Transaction Processing Performance Council. 2024. TPC Benchmark H (TPC-H). http://www.tpc.org/tpch/.

[33] Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. 2023. ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning. *Proc. VLDB Endow.* 16, 11 (2023), 2805–2817.

[34] Shufan Zhang and Xi He. 2023. DProvDB: Differentially Private Query Processing with Multi-Analyst Provenance. *Proc. ACM Manag. Data* 1, 4 (2023), 267:1–267:27.