DP-starJ: A Differential Private Scheme towards Analytical Star-Join Queries

ABSTRACT

Star-join query is the fundamental task in data warehouse and has wide applications in On-line Analytical Processing (OLAP) scenarios. Due to the large number of foreign key constraints and the asymmetric effect in the neighboring instance between the fact and dimension tables, even those latest DP efforts specifically designed for join, if directly applied to star-join query, will suffer from extremely large estimation errors and expensive computational cost.

In this paper, we are thus motivated to propose DP-starJ, a novel **D**ifferentially **P**rivate framework for **star-J**oin queries. DP-starJ consists of a series of strategies tailored to specific features of star-join, including 1) we unveil the different effect of fact and dimension tables on the neighboring database instances, and accordingly revisit the definitions tailored to different cases of star-join; 2) we propose Predicate Mechanism (PM), which utilizes predicate perturbation to inject noise into the join procedure instead of the results; 3) to further boost the robust performance, we propose a DP-compliant star-join algorithm for various types of star-join tasks based on PM. We provide both theoretical analysis and empirical study, which demonstrate the superiority of the proposed methods over the state-of-the-art solutions in terms of accuracy, efficiency, and scalability.

KEYWORDS

star-join, data warehouse, differential privacy, local sensitivity

ACM Reference Format:

. 2024. DP-starJ: A Differential Private Scheme towards Analytical Star-Join Queries. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Technical report).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Star-join query is a common type of query in data warehouse applications, especially on star schema warehouse, where a fact table is joined with one or more dimension tables. It usually performs some filtering on dimension tables, joins the dimension tables with the fact table, and executes some optional aggregation on that. The following provides the formal definition of star-join query.

Definition 1.1 (Star-Join). Let $\mathbf{R}$ be a database schema containing $n+1$ tables, namely $R_0, ..., R_n$. We start with a star-way join :

$$J := R_0(\mathbf{x}_0) \bowtie R_1(\mathbf{x}_1) \bowtie ... \bowtie R_n(\mathbf{x}_n), \quad (1)$$

Technical report,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

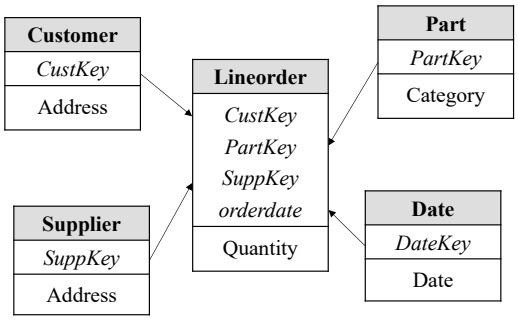

Figure 1: An example of star schema with 4 dimension tables

where $R_1, ..., R_n$ are dimension tables and R_0 is a fact table. We use $[n]$ to denote $\{1, ..., n\}$ and each $\mathbf{x}_i (i \in [n])$ of dimension table $R_i (i \in [n])$ consist of a join key k_i and attribute a_i, $\mathbf{x}_i = \{k_i, a_i\}$. Yet $\mathbf{x}_0$ consist of all join keys k_i and a measure attribute a_0, $\mathbf{x}_0 = \{k_1, ..., k_n, a_0\}$. Let $var(J) := a_0 \cup ... \cup a_n$ is a set of variables in the join result J. Each attribute a_i has a finite domain $dom(a_i)$ with size $|dom(a_i)| = m_i$, the full domain of $\mathbf{R}$ is $dom(\mathbf{R}) = dom(a_0) \times ... \times dom(a_n)$ and has size $m = \prod_i m_i$.

Example 1.2. Suppose a data analyst is interested in the total number of items sold in the first half of this year in a given region, he would execute the following query (assuming the query is performed on the Star Schema Benchmark (SSB) [27]):

```
SELECT count(*)
FROM Date, Customer, Supplier, Part, Lineorder
WHERE Lineroder.CK = Customer.CK
    AND Lineroder.SK = Supplier.SK
    AND Lineroder.PK = Part.PK
    AND Lineorder.orderdate = Date.DK
    AND Customer.region = '[REGION]'
    AND Supplier.region = '[REGION]'
    AND Date.month < 7;
```

Figure 1 shows an example of standard star schema where the query in Definition 1.1 can be applied and Example 1.2 shows an example of a star-join query. Such queries involving star-joins are ubiquitous within analytical tasks and act as a core query category in data warehouse. Due to such a pivotal role in data analytics, star-join has been extensively studied in the literature and widely applied in On-line Analytical Processing (OLAP) practice. Meanwhile, unlike the other types of joint queries that *all tables* can be connected to each other, *all dimension tables* in star-join will be directly linked to the fact table through the foreign-key constraints. In the above example, the relations touched by the query contain private information, *e.g.,* customer c_1 has placed a particular order o_2, suppliers $s_1, \ldots, s_m$ provide an item i_1, of which the privacy must be protected in practical scenarios.

At present, differential privacy (DP) has become a popular solution in privacy-preserving data analytics as it provides a statistically

rigorous privacy guarantee. Since its introduction [10, 11], DP has attracted ever-growing interest in academia, government agencies, and industry. The standard DP mechanism (e.g., *Laplace Mechanism*) first finds the global sensitivity of the query, then it adds a carefully calibrated random noise tailored to the query result. High sensitivity can introduce large noise, which results in a distorted query result offering poor utility. In particular, the global sensitivity of the query refers to how much the query result may change in two neighboring instances of databases. Consequently, a proper definition of neighboring instances is of great importance in DP, which not only determines whether the DP mechanism built on it offers sound and practical privacy protection, but also affects the sensitivity and eventually the utility of the DP mechanism. Since such noise for the privacy-preserving purpose will unavoidably cause utility degradation for the query result, a central problem in DP is how to achieve a satisfactory trade-off between privacy and utility. Existing works [16, 21, 30, 34, 36, 39, 43] have proven that DP mechanism usually achieves a better privacy-utility trade-off when its design is tailored to the specific data analysis task under consideration. In this regard, this paper, for the first time, proposes a solution towards answering star-join queries under differential privacy.

Roughly, the efforts of recent works in DP query processing [8, 18, 35] focus on three aspects to reduce the high global sensitivity: reduce query sensitivity by utilizing the upper bound of the local sensitivity, design an algorithm that effectively computes tight local sensitivity, and transform the database instance by deleting some tuples that are highly sensitive. However, *different from other types of queries that all tables can be linked through join operations, the star-join query has a non-trivial number of foreign key constraints that a single fact table references a series of dimension ones.* Due to that, answering a star-join query in a DP manner is more challenging because high global sensitivity result from the large number of foreign key constraints in star-join query, the DP mechanism of high global sensitivity fails to work, as the output of a join may contain duplicate sensitive rows. This duplication is difficult to bound as it depends on the join type, join condition, and the underlying data. Therefore, the global sensitivity becomes unbounded when joins are present because a single tuple may affect many join results. Therefore, the existing DP-compliant query strategies with trusted server may not be able to provide satisfactory *utility* and *efficiency*, which motivates us to present the solutions in this work.

Example 1.3. The following is a simplest star-join query:
$q := Customer(\underline{CK}, Address, ...) \bowtie Lineorder(CK, orderdate, ...)$.

Here, *Customer* may store customer information and *Lineorder* contains the orders the customers have placed. Then this query simply returns the total numbers of orders. Suppose the identities for the entities in *Customer* are private information we aim to protect. Unfortunately, the global sensitivity of this query is ∞ under existing DP solutions [11]. The reason is as follows, a customer could have an unbounded number of orders, and adding such a customer to the database can cause an unbounded change in the query result theoretically. To address this issue, some works [8, 25] suggest adding data-dependent noise to the query result. For instance, [25] proposed to use the local sensitivity, *i.e.,* the sensitivity of the join query on the given database instance, which is usually much lower than global sensitivity. However, if applied in star-join query, it still leads to high sensitivity and further results in a low utility. The key challenge is how to decrease the global sensitivity of the star-join queries when designing the DP schemes.

Meanwhile, within star-join the tuples from the fact and the dimension table shall affect the query result differently. For this reason, there also exist several different cases for neighboring database instances depending on whether the fact or the dimension tables are private. Therefore, before presenting a well-designed DP-compliant star-join solution, *it is necessary to revisit the definition of neighboring database instances due to the asymmetry between the fact and dimension table.* Accordingly, the DP-compliant star-join solution should take into account the fact that the definition of neighboring database instances may vary between scenarios.

In this paper, we systematically investigate the differential privacy star-join query problem. Our study first reveals that the existing approaches of the traditional DP-compliant join schemes [8, 15, 35], which work by adding subtly noise to the join *result*, fail to achieve a satisfactory utility and efficiency in star-join queries. Thus, we are further motivated to propose an advanced approach called DP-starJ, a **D**ifferentially **P**rivate framework for **star-J**oin queries. To achieve that, we first investigate and unveil the asymmetry between the fact and dimension tables in the effect on neighboring instances of star-join. Driven by that unique nature, instead of considering a uniform definition and simplified case for neighboring database instance as existing DP schemes [7, 8], we propose a fine-grained definition for neighboring database instance tailored to the asymmetry characteristics of star-join task. On the other hand, as discussed in the above, truncating some highly sensitivity tuples or add data-dependent noise towards the result fails to achieve a satisfactory utility and efficiency due to the large number of foreign keys, we are also motivated to propose a new perturbation mechanism to achieve superior utility and efficiency by adding the data-independent noise with bounded global sensitivity, namely Predicate Mechanism. Using the proposed mechanism as building block, we present an DP-compliant star-join algorithm for various types of star-join tasks (*i.e.,* aggregate query, "group_by" operation and workload queries). Further, our theoretical study shows that the proposed methods obtain asymptotically optimal error bound on star-join. Empirical study over several real-world datasets justifies the superiority of our solution in the aspect of both utility and efficiency across various star-join tasks.

The contributions of this paper are summarized as follows:

- We unveil the asymmetry between the fact and dimension tables in affecting the neighboring database instances and revisit the accordingly definitions to tailor to different cases of star-join.
- We proposed a Predicate Mechanism under DP-starJ, which designs a new perturbation strategy to inject noise towards the star-join procedure instead of purely the results. Meanwhile, we further propose an DP-compliant star-join algorithm for various types of star-join tasks.
- We prove theoretically that the proposed method obtains asymptotically optimal error bound on star-join queries and experimental study justifies the superiority of our solution in the aspects of both utility and efficiency.

2 RELATED WORK

Early works mostly focus on answering a given arbitrary SQL queries under DP, which is acknowledged as the holy grail of private query processing. There have been several works on answering various types of queries under DP [2, 5, 17, 23, 41] but not star-join, which has always been the core and basis for the majority of OLAP applications [13]. At present, there is no work that are specifically designed to answer the star-join query in privacy-preserving manner under trusted server settings. Since the elegant work by Dwork [10], there are plenty of works [4, 8, 9, 35] proposed to limit the sensitivity of join queries and extensions for optimizing multi-join queries. In addition, DP-compliant SQL query processing has also been extensively applied in industrial systems, for instance, Uber implements Flex [15] that answers SQL queries with DP.

Many technologies have been proposed to answer set counting queries over a single relation with different predicates [2, 3, 6, 14, 24, 30, 31, 38, 42]. Most existing work on join queries can only support restricted types of joins, such as joins with primary keys [1, 22, 23, 28, 29] and joins with a fixed join attribute [37]. One approach is to reduce the high sensitivity of join queries by truncation. For instance, PrivateSQL [18] uses naive truncation to truncate the tuples with high degrees. Tao *et al.* [35] use naive truncation to truncate the tuples with high sensitivity for some queries without self-join and they propose a mechanism to select the truncation threshold. Dong *et al.* [35] proposed a mechanism Race-to-the-Top (R2T), which can be used to adaptively choose the truncation threshold. However, if applied in star-join, the truncation-based solution will cause significant biased result due to the foreign key constraints between the large number of dimension tables and fact table. Another approach is adding data-dependent noise calibrated by other types of sensitivity rather than global sensitivity. Smooth sensitivity [25] is a popular approach for dealing with multi-way joins. Elastic sensitivity [15] and residual sensitivity [8], both of which are efficiently computable versions of smooth sensitivity, can handle join queries efficiently. However, smooth sensitivity (including any efficiently computable version) cannot support foreign key constraints, which are important to model the relationship between an individual and all his/her associated records in a relational database. Similarly, these methods cannot balance utility and efficiency in the star-join query under DP.

In comparison, the star-join queries have *a non-trivial number of foreign key constraints* in multi-way joins scenarios and the goal is to effectively get accurate query answers even when the star-join query contains a large number of dimension tables. The existing DP-compliant query strategies with trusted server do not satisfy the practical requirements, which motivates us to present the solutions in this work.

3 PRELIMINARIES AND PROBLEM DEFINITION

3.1 Preliminaries

Star-join query

Many relational data warehouse designs today follow a so-called dimensional modeling approach that has been made popular by Galindo *et al.* [13]. Dimensional modeling relies on the distinction of dimension tables with relatively static information in contrast to fact tables that store transactional statistical information. For instance, according to the TPC-H benchmark schema, dimensional table hold master data representing entitles such as *part*, *customers*, *suppliers*, and *date*. In comparison, the fact table in turn stores transactional data, *e.g., lineorder* contains statistics about sales or orders. Dimension tables and fact tables are correlated with each other by foreign key constraints. Usually, fact tables are several orders of magnitude larger than the dimension ones. Dimensional modeling leads to the well-known so-called star schema and snowflake schema design for data warehousing. A star schema consists of a fact table in the center of the star, and it is very popular for modeling data warehouses and data marts. The fact table contains foreign keys, which are pointing to the dimension tables, and the dimension tables contains a key used to joining with the fact table and additional attributes.

Star-join queries are queries on a database instance that the fact table is joined with one or more dimension tables, it selects several measures of interest from the fact table, joins the fact rows with one or several dimensions with respect to the keys, places filter predicates on the business columns of the dimension tables, performs grouping if required, and finally aggregates the measures retrieved from the fact table. As the star-join query in OLAP task places filter predicates on the attributes of the dimension tables, and finally aggregates the measure attribute from the fact table, the star-join query can be converted into predicate query. Predicate queries are a versatile class, consisting of queries that satisfying any logical predicate. A predicate corresponds to condition in the WHERE clause of an SQL statement, and a star-join query is a SQL query with aggregation on measure attributes of fact table and a predicates with equality and range constraints on some dimension tables. The following showcase the template for star-join queries in the form of a standard predicated SELECT SQL statement:

```
SELECT Aggr(*) FROM R WHERE Φ;
```

Aggr(∗) refers to an aggregate function (*e.g.,* COUNT, AVG, SUM) over the fact table. Φ means conjunctions of filter conditions that consists of arbitrary predicates ϕ on attributes over the dimension tables. When a star-join query refers only to an attribute $a_i (i \in [n])$ in dimension table R_i we may say that it is defined with respect to a_i and annotate it as $\Phi := \phi_{a_i}, (\phi_{a_i} : dom(a_i) \rightarrow \{0, 1\})$. Similarly, if ϕ_{a_i} and ϕ_{a_j} are predicates on dimension tables R_i and R_j in a star-join query, then Φ is the conjunctions of predicates $\Phi := \phi_{a_i} \wedge \phi_{a_j}, (\phi_{a_i} \wedge \phi_{a_j} : dom(a_i \cup a_j) \rightarrow \{0, 1\})$.

Let $\mathbf{D}_s$ be a database instance over star schema and a star-join query Q aggregates over the join result $J(\mathbf{D}_s)$. More abstractly, let $\Phi : dom(var(J)) \rightarrow \{0, 1\}$ be an indicator function and the join result satisfy the filter predicate, and $\mathbf{w}(t)$ assigns a non-negative integer weight to the join results only depending on the tuple t. Given the above, we denote the query result of Q on $\mathbf{D}_s$ as $Q(\mathbf{D}_s)$, which can be formally represented as follows.

$$Q(\mathbf{D}_s) = \sum_{t \in J(\mathbf{D}_s)} \Phi(t) \cdot \mathbf{w}(t) \tag{2}$$

Note that the function Φ only depends on the star-join query and t is the tuple in join result $J(\mathbf{D}_s)$. In addition, a star-join query with arbitrary predicate over $var(J)$ can be easily incorporated into this formulation (boolean function): If some $t \in J(\mathbf{D}_s)$ does not satisfy

the predicate, we simply set $\Phi(t) = 0$. For a counting query, **Aggr(∗)** will appear in the form of a COUNT function, $\mathbf{w}(\cdot) = 1$; for other aggregation query, *e.g.,* SUM(a_0), **Aggr(∗)** refers to a SUM function, $\mathbf{w}(t)$ is the value of attribute a_0 for t.

Consider the star-join query towards a database instance in Example 1.2. The query consists of a set of single-table predicates as follows: in the *Date* table, define predicate $\phi_{Date} = \{\mathbb{I}[t_{month} < 7]\}$, $\phi_{Supp} = \{\mathbb{I}[t_{region} = \text{REGION}]\}$ and $\phi_{Cust} = \{\mathbb{I}[t_{region} = \text{REGION}]\}$ in *Supplier* and *Customer* tables. The composite predicate for the query can be expressed as the product $\Phi : \phi_{Date} \wedge \phi_{Cust} \wedge \phi_{Supp}$.

Differential Privacy in Relational Databases with Join Query

Differential Privacy (DP) provides a rigorous privacy guarantee, which has become the de facto privacy-preserving notion in many applications. Before presenting the formal definition of DP, we shall introduce the notion of neighboring database first. For two database instances $\mathbf{D}$ and $\mathbf{D}'$, the distance between $\mathbf{D}$ and $\mathbf{D}'$, denoted $d(\mathbf{D}, \mathbf{D}')$, is the minimum number of steps on which they differ. If $d(\mathbf{D}, \mathbf{D}') = 1$, we call $\mathbf{D}$, $\mathbf{D}'$ neighboring database instances.

Definition 3.1 (Differential Privacy). A randomized algorithm $\mathcal{A}$ satisfies (ϵ, δ) - differential privacy, where $\epsilon, \delta > 0$, if for any pair of neighboring databases $\mathbf{D}$, $\mathbf{D}'$ and any output range $\mathcal{S} \subseteq Range(\mathcal{A})$,

$$Pr[\mathcal{A}(\mathbf{D}) \in \mathcal{S}] \leqslant e^{\epsilon} \cdot Pr[\mathcal{A}(\mathbf{D}') \in \mathcal{S}] + \delta, \tag{3}$$

where the probability is taken over the randomness of $\mathcal{A}$. When $\delta = 0$, it is referred to as pure differential privacy, the algorithm $\mathcal{A}$ satisfies ϵ - differential privacy.

In the above definition, ϵ refers to the privacy budget, which directly restricts the degree for the privacy protection of the algorithm $\mathcal{A}$. Typically, a smaller value of ϵ corresponds to a stronger privacy guarantee. In addition, δ should be much smaller than $1/N_{\mathbf{D}}$ to ensure the privacy of each individual records, where $N_{\mathbf{D}}$ refers to the size of database instance.

Differential privacy is usually achieved by adding random noise drawn from a certain zero-mean probability distribution to the query result. Notably, the magnitude of the random perturbation positively correlates with the difference between the query results on $\mathbf{D}$ and $\mathbf{D}'$, which refers to the notion of sensitivity. The most basic framework of achieving differential privacy is Laplace mechanism, and the noise is scaled according to the *global sensitivity* of the query, defined as follow.

Theorem 3.2 (Laplace Mechanism). *The algorithm* $\mathcal{A}(\mathbf{D}) = Q(\mathbf{D}) + Lap(\frac{GS_Q}{\epsilon})$ *is* ϵ *- differential privacy.*

Definition 3.3 (Global Sensitivity). Let Q denote a particular query, then the global sensitivity of Q, denoted GS_Q, is

$$GS_Q = \max_{\mathbf{D},\mathbf{D}',d(\mathbf{D},\mathbf{D}')=1} \parallel Q(\mathbf{D}) - Q(\mathbf{D}') \parallel . \tag{4}$$

The global sensitivity of the query is defined as the maximal L_1-norm distance between the exact answers of the query Q on any neighboring databases $\mathbf{D}$ and $\mathbf{D}'$. However, unfortunately, the global sensitivity of many queries can be very high. What is worse, for join operator the global sensitivity can be unbounded. Nissim *et al.* [25] proposed a local measure of sensitivity:

Definition 3.4 (Local Sensitivity). For a query Q, the local sensitivity of Q given the database instance $\mathbf{D}$, denoted as $LS_Q(\mathbf{D})$ is as follows:

$$LS_Q(\mathbf{D}) = \max_{\mathbf{D}',d(\mathbf{D},\mathbf{D}')=1} \parallel Q(\mathbf{D}) - Q(\mathbf{D}') \parallel . \tag{5}$$

where the maximum is taken over all neighbors $\mathbf{D}'$ of the particular instance $\mathbf{D}$.

Note that, $GS_Q = \max_{\mathbf{D}} LS_Q(\mathbf{D})$. The local sensitivity is much smaller than global sensitivity in most real-world scenarios. However, an algorithm that releases query result with noise scale proportional to $LS_Q(\mathbf{D})$ on instance $\mathbf{D}$ may not satisfy differential privacy, since $LS_Q(\mathbf{D})$ and $LS_Q(\mathbf{D}')$ can differ a lot on two neighboring instances D and D'. Large difference in the amounts of noise added to $Q(\mathbf{D})$ and $Q(\mathbf{D}')$ may leak sensitive information. To address the issue, Nissim *et al.* [25] proposed the approach that selects noise magnitude according to a smooth upper bound on the local sensitivity instead of using the local sensitivity itself. But differently, compared with the local sensitivity, it is the maximum local sensitivity attained among neighboring instances, the tightest bound is called the *smooth sensitivity*. The smooth sensitivity is based on the *local sensitivity at distance* t, *i.e.,* $LS_Q^{(t)}(\mathbf{D})$, which is defined as

$$LS_Q^{(t)}(\mathbf{D}) = \max_{\mathbf{D}',d(\mathbf{D},\mathbf{D}')\leq t} LS_Q(\mathbf{D}'). \tag{6}$$

Definition 3.5 (Smooth Sensitivity). The β - smooth sensitivity of Q, denoted $SS_Q(\mathbf{D})$, is

$$SS_Q(\mathbf{D}) = \max_{t\geq 0} e^{-\beta t} LS_Q^{(t)}(\mathbf{D}). \tag{7}$$

$SS_Q(\mathbf{D})$ and $SS_Q(\mathbf{D}')$ differ by at most a constant factor on any two neighboring instances $\mathbf{D}$ and $\mathbf{D}'$ to ensure the "smoothness" of $SS_Q(\cdot)$, and the level of smoothness is parameterized by a value β (a smaller value leads to a smooth bound) that depends on ϵ.

3.2 Problem Definition

Differential Privacy in Star-join query

Star-join queries are the most prevalent kind of queries in data warehousing, OLAP and business intelligence applications. Hence, answering star-join query under differential privacy can definitely benefit wide applications in privacy-preserving tasks in the OLAP scenarios. Therefore, in this work we aim to propose the first DP-compliant star-join solution. However, before presenting the solution, due to the special characteristics of the query, it is necessary to reconsider the definition for differential privacy of star-join query. In this subsection, we introduce differential privacy in the star-join query, including neighboring database instances in different situations (fact table and dimension table), and differential privacy in single private relation and multi-private relations with star-join query afterwards.

Consider a database instance $\mathbf{D}_s$ over star schema $\mathbf{R} : \{R_0, R_1, ..., R_n\}$, where R_0 is a fact table and the rest are n dimension tables. Given a star-join query Q shown in Definition 1.1, let $N = |\mathbf{D}_s|$ be the input size, and denote the result of Q on $\mathbf{D}_s$ as $Q(\mathbf{D}_s)$. We consider a DP-compliant star-join based on neighboring instances $\mathbf{D}_s$, $\mathbf{D}_s'$.

Definition 3.6 (Differential Privacy in Star-Join Query). A randomized mechanism $\mathcal{A}$ satisfies ϵ - differential privacy if for neighboring

instances $\mathbf{D}_s$, $\mathbf{D}'_s$ over star-join, where $\epsilon > 0$, and any output range $\mathcal{S} \subseteq Range(\mathcal{A})$,

$$Pr[\mathcal{A}(\mathbf{D}_s) \in \mathcal{S}] \leqslant e^{\epsilon} \cdot Pr[\mathcal{A}(\mathbf{D}'_s) \in \mathcal{S}], \tag{8}$$

where the probability is taken over the randomness of $\mathcal{A}$.

In the above definition, neighboring instances $\mathbf{D}_s$, $\mathbf{D}'_s$ over star schema should differ by one tuple according to the notion of neighboring database. However, in the star schema, each dimension table is independent of each other and has a foreign key constraint referenced by the fact table. The tuples in fact table and dimension tables exert different effects on the query result due to the asymmetric characteristics for both types of tables within the star-join procedure. Therefore, it is necessary to revisit the definition for neighboring instances $\mathbf{D}_s$, $\mathbf{D}'_s$. At the same time, database instance may contain a single private relation or multi-private relations in practical applications. Based on the above reasons, in this subsection we consider the following situations of neighboring database instances $\mathbf{D}_s$, $\mathbf{D}'_s$.

Scenario-dependent Neighboring Database Instance.

As we have discussed above, the unique characteristics for star-join relies in the fact that there exist a large number of foreign key constraints between the fact and dimension tables. As a result, difference in a single tuple within a dimension table may result in a group of different tuples in the fact one. Hence, the asymmetry between both type of tables leads to different scenarios for neighboring instance. In the follow, we shall discuss accordingly.

Definition 3.7 ((a,b)-private). Given the star-join task shown in Definition 1.1, which contains at least one sensitive table, we refer to the scenario as (a,b)-**private** if a number of a ($a \in \{0,1\}$) fact tables and b ($b \leq n, a+b \geq 1$) dimension ones are sensitive.

$(1)(0,k)$*-private.* The private relations are all dimension tables, $R_p^1, ..., R_p^k \in \{R_i\}(i \in [n], k \leq n)$. When the database instance $\mathbf{D}_s$ exist the foreign key constraint that table has foreign key referencing the primary key (PK) of the other table, the two instances $\mathbf{D}_s$ and $\mathbf{D}'_s$ are considered as neighbors if $\mathbf{D}'_s$ can be obtained from $\mathbf{D}_s$ by: deleting a tuple t from the referenced table, and a set of tuples that reference t in the referencing table. As each dimension table has a foreign key constraint with the fact table, we adopt the DP policy in star-join query, which defines neighboring instances by taking foreign key constraints into consideration. The basic private relation of $(0,k)$-private is to only include one dimension table, that is, when $k = 1$. Therefore, we refer to $\mathbf{D}_s$, $\mathbf{D}'_s$ as neighboring instances over star schema if all tuples in the difference between $\mathbf{D}_s$ and $\mathbf{D}'_s$ reference a single tuple t_p in the private dimension table R_p^k. In particular, $t_p (t_p \in R_p^k)$ may also be deleted, in which case all tuples referencing t_p in the fact table must be deleted in order to preserve the foreign key constraints. When $k > 1$, since each dimension table is independent of each other and the fact table has foreign keys referencing the primary key of each dimension one, thus we assign unique identifiers to the conjunction of all foreign keys in the fact table. If $\mathbf{D}'_s$ can be obtained form $\mathbf{D}_s$ by deleting a tuple $t_p^i \in R_p^i$ for each private relations, as well as all the tuples in the fact table referencing the same tuple $t \in t_p^1(PK) \wedge ... \wedge t_p^k(PK)$, we call $\mathbf{D}_s$, $\mathbf{D}'_s$ neighboring instances in this case.

$(2)(1,k)$*-private.* The private relations contains the fact table. The simplest scenario of $(1,k)$-private is $k = 0$, which means that only the fact table is private. When $k = 0$, two instances can only differ at one tuple in the fact table, $\mathbf{D_s}$, $\mathbf{D}'_s$ are referred to as neighboring instances, $d(\mathbf{D}_s, \mathbf{D}'_s) = 1$. Another scenario of $(1,k)$-private is the case when $k \neq 0$, *i.e.,* some of the dimension tables are private. In this case, two neighboring instances $\mathbf{D}_s$, $\mathbf{D}'_s$, can differ at one tuple in the fact table. Moreover, similar to $(0,k)$-private, $\mathbf{D}'_s$ also needs to be obtained form $\mathbf{D}_s$ by deleting a tuple $t_p^i \in R_p^i$ from each private dimension tables, as well as all the tuples in the fact table referencing the same tuple $t \in t_p^1(PK) \wedge ... \wedge t_p^k(PK)$.

The above outlines the different cases for neighboring instances $\mathbf{D}_s$, $\mathbf{D}'_s$ in the star-join query. In Definition 1.1, star-join queries are transformed to predicate queries in the multidimensional data cube. Therefore, the algorithm that satisfies differential privacy is implemented for each predicate constraint of the star-join query Q, so that the query Q conforms to differential privacy.

4 BASIC MECHANISM FOR STAR-JOIN QUERY: OUTPUT PERTURBATION

In order to systematically find the ideal solution for answering star-join query under DP, we investigate ways through both the output and input perturbations. In this section, we propose the basic approach for DP-compliant star-join query by a pair of output-based perturbation mechanisms. Aside with that, we also conduct theoretical utility study, which shows that the basic mechanism achieves satisfactory (although not elegant) trade-off between utility, efficiency and scalability.

Intuitively, following the standard DP solutions, we can propose a basic strategy by approximating real-valued functions based on adding a small amount of random noise to the true answer. In particular, we introduce both a data-independent approach and a data-dependent one to the star-join query result according to whether the global sensitivity of star-join query is bounded. In data-independent approach, if the global sensitivity of star-join query is bounded, the server is in charge of adding random noise to the query result. The most popular method is to rely on Laplace Mechanism (LM) that scales according to the global sensitivity GS_Q of the star-join query Q. The variance of the Laplace Mechanism is $2(\frac{GS_Q}{\epsilon})^2$.

In star-join query, this method is only applicable with the $(1,0)$-private scenario, where the fact table is the only one that is sensitive. Besides that, Laplace mechanism will fail to work in the $(\cdot,k)$-private relation contains dimension table due to the unbounded global sensitivity. Notably, in practical scenarios, sensitive information is mostly contained in the dimension tables rather than the fact one (*e.g., Customer* is a private relation that need to be protected).

For those cases when private relation includes dimension table, that is, the global sensitivity is unbounded, we first consider to adopt a data-independent approach by utilizing Truncation Mechanism (TM) that bounds the global sensitivity by simply deleting all records, the sensitivity of which is larger than a predefined threshold τ, before adding random noise to the true answer. However, a well-known limitation for the truncation mechanism is the bias-variance trade-off: a large threshold τ will lead to large random noise with tremendous variance; while a small τ may introduces a

bias as large as the query result itself. When the private relation contains dimension table, due to the aforementioned limitation of LM and TM in data-independent approach, we select to adopt data-dependent approach by injecting data-dependent noise into the query result.

The data-dependent approach involves applying Local Sensitivity (LS) and Race-to-the-Top (R2T) to the star-join query. The LS is usually a two-phase strategy as follows.

(1) compute the upper bound of local sensitivity $\hat{LS}_Q(\mathbf{D}_s)$ in star-join query Q with database instance $\mathbf{D}_s$;
(2) add the noise that calibrates the size of $\hat{LS}_Q(\mathbf{D}_s)$ to the query result.

In general, there are two mechanisms for implementing LS, namely Cauchy Mechanism and Laplace Mechanism. Cauchy Mechanism works by setting $\beta = \frac{\epsilon}{2(\gamma+1)}$, and then adds noise $Cauchy(\frac{\hat{LS}_Q(\mathbf{D}_s)}{\beta})$ to the query answer $Q(\mathbf{D}_s)$. It preserves ϵ - differential privacy, where $Cauchy(\cdot)$ is drawn from the general Cauchy distribution. For instance, suppose we set $\gamma = 4$ for which $Var(Cauchy(\cdot)) = 1$, and the noise level of Cauchy Mechanism is thus $(\frac{10\hat{LS}_Q(\mathbf{D}_s)}{\epsilon})^2$. Notably, as there is a long tail in the general Cauchy distribution, which decays only polynomially comparing with the exponential decay within the Laplace distribution, one can use the Laplace distribution to achieve a better concentration. However, the Laplace Mechanism only yield (ϵ, δ) - differential privacy. The Laplace Mechanism works by setting $\beta = \frac{\epsilon}{2ln(\frac{2}{\delta})}$, and adds noise $Lap(\frac{2\hat{LS}_Q(\mathbf{D}_s)}{\epsilon})$ to the true answer $Q(\mathbf{D}_s)$. Since $Var(Lap(\cdot)) = 2$, the noise level of Laplace Mechanism is $8(\frac{\hat{LS}_Q(\mathbf{D}_s)}{\epsilon})^2$.

Another method in data-dependent approach is Race-to-the-Top (R2T). It is a truncation mechanism with foreign key constraints in join query, and can be used in combination with any truncation method. The basic idea of R2T is to try geometrically increasing values of truncation threshold τ and somehow pick the "winner" from all the trials. The R2T first computes the query result $Q(\mathbf{D}_s, \tau)$ with various threshold τ, and then adds $Lap(\frac{\tau}{\epsilon})$ to $Q(\mathbf{D}_s, \tau)$ to get the noise result $\hat{Q}(\mathbf{D}_s, \tau)$, which would turn it into an ϵ - differential privacy mechanism. Finally, returning the maximum $\hat{Q}(\mathbf{D}_s, \tau)$ preserves DP by the post-processing property of differential privacy. The R2T works as follows:

For $\tau^{(j)}$, $j = 1, ..., log(GS_Q)$,

$$\hat{Q}(\mathbf{D}_s, \tau^{(j)}) = Q(\mathbf{D}_s, \tau^{(j)}) + Lap(log(GS_Q)\frac{\tau^{(j)}}{\epsilon}) \\ -log(GS_Q)ln(\frac{log(GS_Q}{\alpha}) \cdot \frac{\tau^{(j)}}{\epsilon}, \tag{9}$$

and then outputs $max\{max_j\hat{Q}(\mathbf{D}_s, \tau^{(j)}), Q(\mathbf{D}_s, 0)\}$, where α is the probability concern about the utility. The R2T mechanism satisfies ϵ - differential privacy by the basic composition theorem [12]. Note that, $Q(\mathbf{D}_s, \tau)$ is different in queries with and without self-join, it may rely on Linear Program(LP)-based truncation mechanism when there exists self-join in the query. For the utility of R2T, we have $Q(\mathbf{D}_s) - 4log(GS_Q)ln(\frac{log(GS_Q}{\alpha})\frac{\tau^*(\mathbf{D}_s)}{\epsilon} \leq \hat{Q}(\mathbf{D}_s)$ with probability at least $1-\alpha$. Hereby $\tau^*(\mathbf{D}_s)$ means a bound of threshold that holds for any $\tau \geq \tau^*(\mathbf{D}_s)$, $Q(\mathbf{D}_s, \tau) = Q(\mathbf{D}_s)$.

Remark 1. In the star-join query, the sensitivity of query plays an important role on the output mechanism. From the aspect of the output perturbation, the utility is directly affected by the noise that scaled according to the sensitivity of star-join query. Both the global and the local sensitivity are extremely high, due to the existence of join operations in star-join query. Especially for an n-way star join, the global sensitivity can be as high as $O(N^{n-1})$, which is unbounded as $N = |\mathbf{D}_s|$ is the input size. Therefore, this brings down the utility because the GS_Q of the star-join query can be ∞ under pure DP. Although using the instance-depended noise, the output mechanism has the intrinsic limitation on achieving high utility due to fact that the smooth upper bound of $LS_Q(\mathbf{D}_s)$ is very large in practical applications, and the computational cost of it is extremely high. In short, the high sensitivity of star-join query result extremely limits the utility level that the basic output perturbation mechanism can achieve.

Remark 2. Although the output mechanism adopts the smooth sensitivity to reduce the noise for better utility, in fact, it is shown that for certain problems, computing or even approximating the smooth sensitivity is NP-hard [35]. Therefore, the computational hardness of smooth sensitivity of star-join queries increases with the increase of multi-way joins. [8] argues that it may not be NP-hard, and even if there is a polynomial-time algorithm to compute the smooth sensitivity, it will be inevitably too complicated in practice. Thus it is challenging for the output perturbation mechanism to achieve satisfactory scalability and is impractical in realistic scenarios.

5 ADVANCED APPROACH: DP-STARJ

Motivated by the limitation in achieving an elegant tradeoff between utility, efficiency and scalability under the output mechanism, we propose an advanced approach of DP-starJ, which can achieve strict DP with higher utility and efficiency to answer the star-join query. The main idea of DP-starJ is decomposing high sensitivity using the intrinsic characteristics of star-join to balance the utility and efficiency. Compared with the output perturbation mechanism, DP-starJ avoids the high sensitivity of star-join queries while improving the utility and reducing the computation cost.

The overall intuition of DP-starJ is to add noise to star-join queries from the view of input, which turns out to be a challenging task. In the following, we first present a framework of DP-starJ to answer star-join query under DP and then propose a mechanism of input perturbation in DP-starJ, namely, Predicate Mechanism (PM). Afterwards, we introduce DP-starJ to support for various types of star-join queries. At last, we give the granularity privacy and utility study.

5.1 DP-starJ

As discussed in Section 4, none of the existing mechanisms can overcome all three key challenges (utility, efficiency and scalability) in DP-compliant star-join query. To address this problem, we first propose a framework called DP-starJ that answers the star-join query under DP. It's main idea is to add random noise to star-join query procedure rather than the query result, DP-starJ decomposes the predicates of star-join query to reduce the high global sensitivity of the query. Specifically, DP-starJ mainly consists of three phases as shown in Figure 2:

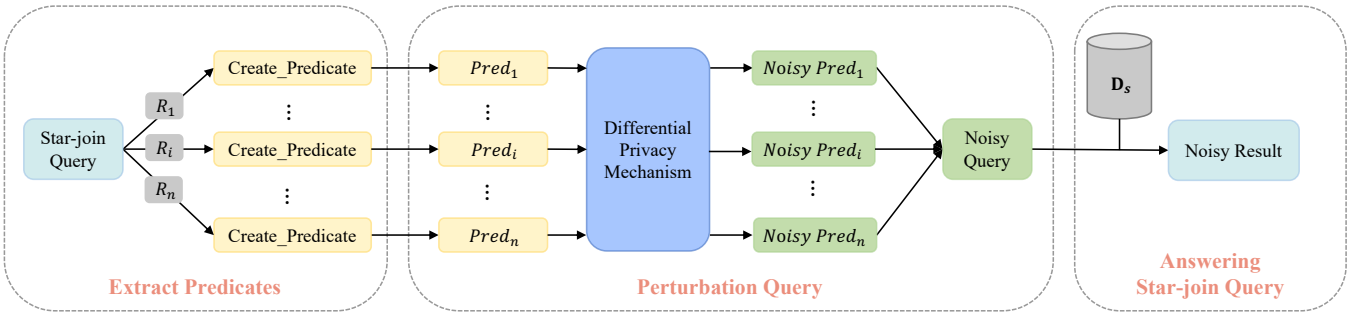

Figure 2: Execution Phases with DP-starJ

Phase 1. Extract Predicates. In DP-starJ, given the star-join query Q with n dimension tables and a fact table R_0, the server first extracts predicates from the query. The star-join query can be expressed as a predicate query since the star structure of database instance and the independence of dimension tables. Therefore, in this phase, the server mainly extracts predicates of each dimension table according to the star-join query and database schema. In star-join query, the type of predicate typically includes the range constraint and point constraint of attribute in the dimension table. Hence, the phase extracts predicates based on the dimension table involved in the given query. If the star-join query includes all dimension tables, the server will create n predicates for each of n dimension tables, respectively.

Phase 2. Perturbation Query. In this phase, we employ some perturbation mechanisms to the star-join query to ensure differential privacy of DP-starJ framework. The main process is to add random noise towards the predicates generated in Phase 1. Afterwards, it aggregates all noise predicates together into noised star-join query, where the perturbation mechanism adopted is orthogonal and various specific methods can be employed, *i.e.,* in this paper we use the Laplace noise for each attribute.

Phase 3. Answering Star-join Query. In this phase, the server answers the star-join query Q in a DP manner by accessing the database instance $\mathbf{D}_s$ with the noised star-join query $\hat{Q}$.

To balance the utility, efficiency and scalability, DP-starJ responds to the star-join query in the form of an input perturbation. In addition, it decomposes predicates to reduce the sensitivity of query in order to improve the utility. In the following, we present Predicate Mechanism to implement the DP-starJ framework, which helps us identify the key problems for developing DP-starJ.

5.2 Predicate Mechanism

Let $\mathbf{D}_s$ be a database instance over star schema and a star-join query Q aggregates over the join result $J(\mathbf{D}_s)$. Since Φ is an indicate function, we simplify Equation 2 to the following form:

$$Q(\mathbf{D}_s) = \sum_{t \in J(\mathbf{D}_s)} \Phi(t) \cdot \mathbf{w}(t) = \mathbf{\Phi} \cdot \mathbf{w}(t) \tag{10}$$

Hereby $\mathbf{\Phi}$ refers to a predicate matrix of star-join query with all record. Since in star-join queries each dimension table $R_i, i \in [n]$ is independent of each other and places filter predicates towards the attributes locally, thus $\mathbf{\Phi}$ can reflect the conjunctions of predicate, $\Phi := \phi_{a_1} \wedge \phi_{a_2} \wedge ... \wedge \phi_{a_n}$. Moreover, we can vectorize the weight function of tuple $\mathbf{w}(t)$ as $\mathbf{W}$, so the above equation can be transformed into the follows:

$$Q(\mathbf{D}_s) = \Phi \cdot \mathbf{w}(t) = \Phi \cdot \mathbf{W} = (\phi_{a_1} \wedge \phi_{a_2} \wedge ... \wedge \phi_{a_n}) \cdot \mathbf{W} \tag{11}$$

where ϕ_{a_i} is the predicate condition of dimension table R_i in the star-join query Q.

Algorithm 1: Predicate Mechanism

Input: Star-join query Q, Data instance $\mathbf{D}_s$, Data Matrix $\mathbf{W}$, parameter ϵ
Output: Noisy result: $\hat{Q}(\mathbf{D}_s)$

$\Phi \leftarrow Q$;
$\epsilon_i = \frac{\epsilon}{n}$;
for *each predicate* $\phi_{a_i} \in \Phi$ **do**
  $\hat{\phi}_{a_i} \leftarrow \phi_{a_i} + Lap(\frac{dom(a_i)}{\epsilon_i})$
$\hat{\Phi} \leftarrow \hat{\phi}_{a_1} \wedge ... \wedge \hat{\phi}_{a_n}$;
$\hat{Q}(\mathbf{D}_s) = \hat{\Phi} \cdot \mathbf{W}$;
Return $\hat{Q}(\mathbf{D}_s)$

Unlike the output perturbation, the Predicate Mechanism adds random perturbation to the predicates involved in the star-join query procedure before touching the raw database instance.

$$\begin{aligned}\hat{Q}(\mathbf{D}_s) &= (\Phi + Lap(\frac{GS_\Phi}{\epsilon})) \cdot \mathbf{W} \\ &= ((\phi_{a_1} + Lap(\frac{GS_{\phi_{a_1}}}{\epsilon_1})) \wedge ... \wedge (\phi_{a_n} + Lap(\frac{GS_{\phi_{an}}}{\epsilon_n}))) \cdot \mathbf{W} \\ &= (\hat{\phi}_{a_1} \wedge ... \wedge \hat{\phi}_{a_n}) \cdot \mathbf{W} \\ &= \hat{\Phi} \cdot \mathbf{W}\end{aligned} \tag{12}$$

where the privacy cost is $\epsilon_i = \frac{\epsilon}{n}$, and the global sensitivity $GS_{\phi_{a_i}}$ of each predicate ϕ_{a_i} is the domain size of attribute a_i in dimension table R_i. Algorithm 1 shows the pseudo-code of PM, the server (i) generates the predicate Φ in a star-join query Q, (ii) decomposes Φ into dimension table predicates ϕ_{a_i} based on Q and adds noise to the predicates ϕ_{a_i}, and (iii) answers star-join query Q according to the noised predicate $\hat{\Phi}$ and finally obtains the noise result $\hat{Q}(\mathbf{D}_s)$. The main idea of PM is to add random noise to each predicate ϕ_{a_i} of dimension tables in star-join query Q because the predicates of each dimension table are independent of each other. We now carry on with the predicate perturbation of each single dimension table for predicate mechanism implementation.

Algorithm 2: Predicate Mechanism for An Attribute ($\mathbf{PM_A}$)

Input: Predicate ϕ_{a_i} of an attribute a_i, parameter ϵ
Output: Noisy Predicate: $\hat{\phi}_{a_i}$

if ϕ_{a_i} *is* $a_i = v$ **then**
  $\hat{v} = v + Lap(dom(a_i)/\epsilon)$;
  $\hat{\phi}_{a_i} \leftarrow a_i = \hat{v}$
else
  $\phi_{a_i} \leftarrow a_i \in [l, r]$;
  $\hat{l} = l + Lap(\frac{2 \cdot dom(a_i)}{\epsilon})$;
  $\hat{r} = r + Lap(\frac{2 \cdot dom(a_i)}{\epsilon})$;
  while $\hat{l} < \hat{r}$ **do**
    $\hat{\phi}_{a_i} \leftarrow a_i \in [\hat{l}, \hat{r}]$
Return $\hat{\phi}_{a_i}$

In the predicate perturbation, a straightforward solution is to perturb each predicate separately using a single Laplace perturbation algorithm, such that every attribute is given a privacy budget $\epsilon_i = \epsilon/n$. Then, it is well known that the Laplace perturbation is suitable for a real-value, the predicate of query may contain point constraints and range constraints of an attribute. For the two classes of predicates, we use Laplace noise to perturb predicates with point constraints and range constraints, respectively. The specific process is as follows.

Predicate Perturbation for Each Single Attribute. In an attribute a_i, the predicate ϕ_{a_i} in dimension table R_i may contain either range constraints $a_i \in [l, r]$, or point constraints $a_i = v$. If the predicate is a point constraints, the predicate perturbation is directly adding the Laplace noise to the value v. When the the predicate is a range constraints, $a_i \in [l, r]$, the predicate perturbation is to perturb both ends of the interval $[l, r]$ independently using a Laplace perturbation algorithm, such that every attribute is given a privacy budget $\epsilon/2$. The specific process is shown in Algorithm 2.

5.3 DP-starJ Applications

To further boost the robust performance, in this section, we discuss specific solutions for predicate mechanism in DP-starJ for various types of star-join tasks. The main idea of DP-starJ is to inject random data-independent noise to star-join query, which the application of predicate mechanism on different star-join queries. Therefore, we present the predicate mechanism for aggregated star-join queries, "Group_By" operation, and star-join workload queries as follow.

Predicate Mechanism for Aggregated Star-join Queries. We now consider the case for the star-join aggregation query that aggregates the number of tuples that suit for the filter conditions. In this case, the solution is to perturb each predicate independently using a single predicate perturbation algorithm (Algorithm 2), such that every attribute is given a privacy budget $\epsilon_i = \epsilon/n$. The specific process is shown in Algorithm 3, where the data matrix is **1** in which the value of all tuples is 1. If the aggregation function is the SUM in the star-join, the element of data matrix is the value of attribute, which is the summation over the attributes in the star-join query. In addition, if the star-join query involves "Group_By" operation, similar to COUNT queries and SUM queries, we shall only perturb the predicates of the query before "Group_By" operation. Therefore,

Algorithm 3: Predicate Mechanism for Star-join Counting Query

Input: Star-join counting query Q_c, Data instance $\mathbf{D}_s$, Data Matrix $\mathbf{W}$, parameter ϵ
Output: Noisy result: $\hat{Q}_c(\mathbf{D}_s)$

$\Phi \leftarrow Q_c$;
$\epsilon_i = \frac{\epsilon}{n}$;
for *each predicate* $\phi_{a_i} \in \Phi$ **do**
  $\hat{\phi}_{a_i} \leftarrow \mathbf{PM_A}(\phi_{a_i}, \epsilon_i)$
$\hat{\Phi} \leftarrow \hat{\phi}_{a_1} \wedge ... \wedge \hat{\phi}_{a_n}$;
$\hat{Q}_c(\mathbf{D}_s) = \hat{\Phi} \cdot \mathbf{W}$;
Return $\hat{Q}_c(\mathbf{D}_s)$

DP-StarJ supports not only ordinary aggregate queries but also "Group_By" statement in star-joins.

Predicate Mechanism for Star-join Workload Queries. In addition, as workload task are ubiquitous in OLAP scenarios [32], we extensively consider answering star-join workload queries under differential privacy by using PM. Given a workload of l star-join queries $\mathbf{L}$, $\mathbf{L} = \{Q_1, Q_2, ..., Q_l\}$. One straightforward solution is to process each query Q_i independently by using the Predicate Mechanism. Unfortunately, this strategy fails to exploit the correlations between different queries, which has been exhaustively studied and justified to be valuable in designing a more effective DP solution [20, 40]. Consider a workload of three different queries, Q_1 is interested in the total number of products sold in the first half of this year, while Q_2 is interested in the total number of products sold in the second half of this year, and Q_3 asks for the total number throughout the whole year. Clearly, the three queries are correlated with each other as $Q_3 = Q_1 + Q_2$. Given that fact, an alternative strategy for answering these queries is to process only Q_1 and Q_2, and use their sum to answer Q_3. Inspired by this phenomenon, we propose a Workload Decomposition (WD) strategy to answer star-join workload queries under differential privacy in the following.

Consider the star-join workload queries $\mathbf{L} = \{Q_1, Q_2, ..., Q_l\}$. According to our discussion in Section 3, each star-join query Q_i can be represented by its predicate Φ_i. Following that way, the star-join workload queries $\mathbf{L}$ can be accordingly represented as a set of predicates Φ_i, $\mathbf{L} := \{\Phi_1, \Phi_2, ..., \Phi_l\}$. Each predicate Φ_1 refers to filter conditions for different dimension tables, $\Phi_i := \phi^i_{a_1} \wedge \phi^i_{a_2} \wedge ... \wedge \phi^i_{a_n}$.

Firstly, we adopt one-hot-encoding to quantify Φ_i into a series of vectors. As shown in Example 1.2, the predicate of the star-join query is $\Phi = \phi_{Date} \wedge \phi_{Cust} \wedge \phi_{Supp}$, assume that the domain of *region* is {A, B, C} and REGION = C, we can vectorized Φ as [111111000000001001]. Similarly, the vector representation for ϕ_{Date} and $\phi_{Cust}(\phi_{Supp})$ are [111111000000] and [001]([001]), respectively. Therefore, the workload queries $\mathbf{L}$, *i.e.,* a collection of l star-join queries, can be arranged by rows and forms an $l \times m_d$ matrix and $m_d = \prod_{i=1}^{n} m_i$. The predicate matrix $\mathbf{P}^{\mathbf{L}}_i$ of each dimension table R_i on workload queries $\mathbf{L}$ is an $l \times m_i$ matrix, hereby m_i is the domain size of attribute on dimension table R_i.

Secondly, for each predicate matrix $\mathbf{P}^{\mathbf{L}}_i$, we shall perform a matrix decomposition as follow:

Definition 5.1 (Matrix Decomposition). Given a predicate matrix $\mathbf{M}$ and a strategy matrix $\mathbf{A}$, we say $\mathbf{M}$ decomposes into $\mathbf{XA}$ if each predicate in $\mathbf{M}$ can be expressed as a linear combination of predicates in $\mathbf{A}$. In other words, there exists a solution matrix $\mathbf{X}$ to $\mathbf{M} = \mathbf{XA}$.

For each predicate matrix on the workload queries $\mathbf{L}$, $\mathbf{P}_i^{\mathbf{L}}$, MD shall finds a new strategy matrix $\mathbf{A}_i$ to support $\mathbf{P}_i^{\mathbf{L}}$, and then evaluates the strategy matrix $\mathbf{A}_i$ using the Predicate Mechanism to obtain a noisy strategy matrix $\hat{\mathbf{A}}_i$. Afterwards, we can reconstruct noisy predicate matrix from the noisy strategy matrix $\hat{\mathbf{A}}_i$, $\hat{\mathbf{P}}_i^{\mathbf{L}} = \mathbf{A}_i^{+}\hat{\mathbf{A}}_i$.

Finally, we connect the noisy predicate matrix $\hat{\mathbf{P}}_i^{\mathbf{L}}$ to each corresponding dimension table into the noisy predicate matrix $\hat{\mathbf{P}}$. The server answers the star-join workload queries $\hat{\mathbf{L}}$ in a DP manner by accessing the database instance with the noisy predicate matrix $\hat{\mathbf{P}}$ on the noisy star-join workload queries $\hat{\mathbf{L}}$.

Algorithm 4 outlines the above procedure. It first uses one-hot-encoding to represent the predicate of star-join the workload queries $\mathbf{L}$ and assigns the privacy budget to the predicate matrix that decomposes the predicate according to the dimension table (Lines 1-2). After that, matrix decomposition is performed on each predicate matrix $\mathbf{P}_i^{\mathbf{L}}$ to get the corresponding strategy matrix $\mathbf{A}_i$ and applies Predicate Mechanism on the strategy matrix to obtain a noised strategy matrix $\hat{\mathbf{A}}_i$, and reconstruct the noise-injected predicate matrix $\hat{\mathbf{P}}_i^{\mathbf{L}}$ through the noise-injected strategy matrix (Lines 3-9). Afterwards, it connects each noise-injected predicate matrix $\hat{\mathbf{P}}_i^{\mathbf{L}}$ into $\hat{\mathbf{P}}$ of the noisy star-join workload queries $\hat{\mathbf{L}}$ and obtain the noisy result $\hat{Q}_L(\mathbf{D}_s)$ of the workload by accessing the database instance $\mathbf{D}_s$ (Lines 10-12), and finally outputs the noisy result $\hat{Q}_L(\mathbf{D}_s)$.

Predicate Mechanism for snowflake queries. Besides star-join, the mechanism can also be applied to snowflake model (*resp.*, snowflake query), which further hierarchizing the dimension tables of the star schema, resulting in a more normalized structure. For example, in Figure 1 and Example 1.2, *Date* can be decomposed into dimension tables such as Year, Quarter, Month, and Day, reducing redundancy. Therefore, the star-join query in Example 1.2 can be extended to snowflake query by changing $Date.month < 7$ to $Date.MK = Month.MK\ AND\ Month.month < 7$. At this point, we can still use predicate mechanism to perturb the predicate and obtain the perturbed snowflake query. This does not affect the functionality of DP-starJ though extending star-join queries to queries on snowflake model.

5.4 Theoretical Study over the Privacy and Utility

We now conduct theoretical study over the privacy guarantee as well as the utility for the proposed Predicate Mechanism. In this section, we first study the privacy guarantee of Predicate Mechanism and DP-starJ. After that, we theoretically study the utility of Predicate Mechanism.

Theorem 5.2. *Algorithm 2 satisfies ϵ-differential privacy.*

Proof. Algorithm 2 in the paper adds Laplace noise to the predicate, and the scale of Laplace is the ratio of the domain and privacy cost. In the worst case, the number of ways a change in a record can affect the predicate is equal to the size of the domain. Therefore, its global sensitivity is the size of the domain. In other words,

Algorithm 4: Predicate Mechanism for Star-join Workload Queries

Input: Star-join Workload Queries $\mathbf{L} = \{Q_1, Q_2, ..., Q_l\}$, Data instance $\mathbf{D}_s$, Data Matrix $\mathbf{W}$, parameter ϵ
Output: Noisy result: $\hat{Q}_L(\mathbf{D}_s)$

1 $\mathbf{P} \leftarrow \mathbf{One-Hot-Encoding}(\mathbf{L})$;
2 $\epsilon_i = \frac{\epsilon}{n}$;
3 **for** *each predicate matrix* $\mathbf{P}_i^{\mathbf{L}} \in \mathbf{P}$ **do**
4 $\quad \mathbf{A}_i \leftarrow \mathbf{MatrixDecom}(\mathbf{P}_i^{\mathbf{L}})$;
5 $\quad \phi_{a_i} \leftarrow \mathbf{A}_i$;
6 $\quad \hat{\phi}_{a_i} \leftarrow \mathbf{PM}_{\mathbf{A}}(\phi_{a_i}, \epsilon_i)$;
7 $\quad \hat{\mathbf{A}}_i \leftarrow \hat{\phi}_{a_i}$;
8 $\quad \hat{\mathbf{P}}_i^{\mathbf{L}} = \mathbf{A}_i^{+}\hat{\mathbf{A}}_i$;
9 $\hat{\mathbf{P}} \leftarrow \hat{\mathbf{P}}_1^{\mathbf{L}}, ..., \hat{\mathbf{P}}_n^{\mathbf{L}}$;
10 $\hat{\mathbf{L}} \leftarrow \hat{\mathbf{P}}$;
11 $\hat{Q}_L(\mathbf{D}_s) = \hat{\mathbf{L}} \cdot \mathbf{W}$;
12 Return $\hat{Q}_L(\mathbf{D}_s)$

Algorithm 2 essentially implements the Laplace mechanism on the predicate. According to the Theorem 3.2, Algorithm 2 satisfies ϵ-differential privacy. □

Theorem 5.3. *Predicate Mechanism and DP-starJ satisfy ϵ-differential privacy.*

Proof. The Predicate Mechanism decomposes Φ into dimension table predicates ϕ_{a_i} based on Q and adds noise to the predicates ϕ_{a_i}, the proof of PM is transformed into proving that each predicates $\hat{\phi}_{a_i}$ satisfies $\frac{\epsilon}{n}$-differential privacy according to the Theorem 5.2, and whether the predicate $\hat{\Phi}$ satisfies ϵ-differential privacy. According to the fact that $\Phi := \phi_{a_1} \wedge \cdots \wedge \phi_{a_1}$ and each ϕ_{a_1} is independent of each other, thus, $Pr[\Phi] = Pr[\phi_{a_1} \wedge \cdots \wedge \phi_{a_1}] = Pr[\phi_{a_1}] \cdot Pr[\phi_{a_2}] ... Pr[\phi_{a_n}]$.
Meanwhile, each predicates ϕ_{a_i} satisfies $\frac{\epsilon}{n}$-differential privacy, $Pr[\phi_{a_i}] \leq e^{\frac{\epsilon}{n}} \cdot Pr[\hat{\phi}_{a_i}]$. Therefore,

$$\begin{aligned} Pr[\Phi] &= Pr[\phi_{a_1}] \cdot Pr[\phi_{a_2}] \cdots Pr[\phi_{a_n}] \\ &\leq e^{\epsilon} \cdot (Pr[\hat{\phi}_{a_1}] \cdot Pr[\hat{\phi}_{a_2}] \cdots Pr[\hat{\phi}_{a_n}]) = e^{\epsilon} \cdot Pr[\hat{\Phi}] \end{aligned} \tag{13}$$

The Predicate Mechanism satisfies ϵ-differential privacy. Similar to Predicate Mechanism, DP-starJ can be proved to satisfy ϵ-differential privacy in the same way. □

Both of Algorithm 3 and Algorithm 4 adopt the Predicate Mechanism, we shall study the privacy guarantee of them accordingly as follows.

Theorem 5.4. *Algorithm 3 satisfies ϵ-differential privacy.*

Proof. Algorithm 3 decomposes query predicates by dimension tables, allocating the privacy budget of $\frac{\epsilon}{n}$ to each predicate. According to the Theorem 5.2, each noisy predicate satisfies $\frac{\epsilon}{n}$-differential privacy via Algorithm 2. Within each predicate there is sequential composition because adding or removing a record affects all predicates. According to Sequential Composition [12], Algorithm 3 satisfies ϵ-differential privacy. □

Theorem 5.5. *Algorithm 4 satisfies ϵ-differential privacy.*

Table 1: Relative error(%) of various mechanisms PM, R2T, LS on SSB queries by varying ϵ.

Query type		COUNT				SUM			GROUP BY	
Query		Q_{c1}	Q_{c2}	Q_{c3}	Q_{c4}	Q_{s2}	Q_{s3}	Q_{s4}	Q_{g2}	Q_{g4}
$\epsilon = 0.1$	PM	11.89	9.46	19.02	8.22	12.07	16.3	17.36	11	28.63
	R2T	120.87	41.51	29.63	20.41	80.61	80.22	80.14	Not supported*	
	LS	180.9	73.39	78.12	80.44	Not supported				
$\epsilon = 0.2$	PM	11.93	9.28	16.48	5.12	11.55	13.07	12.39	10.6	18.8
	R2T	59.76	30.38	19.4	15.16	79.91	80.17	79.83	Not supported*	
	LS	121.68	61.8	58.61	83	Not supported				
$\epsilon = 0.5$	PM	8.66	7.61	15.42	4.3	11.58	12.45	10.43	9.88	11.83
	R2T	84.48	22.9	19.67	11.55	79.46	80.08	79.61	Not supported*	
	LS	86.84	47.6	20.38	52.09	Not supported				
$\epsilon = 0.8$	PM	5.1	7.86	13.35	3.71	11.43	12.59	7.58	9.25	6.45
	R2T	76.16	17.46	14.56	9.21	79.21	80.03	79.17	Not supported*	
	LS	77.23	32.99	13.28	31.89	Not supported				
$\epsilon = 1$	PM	5	7.53	11.76	1.92	10.51	12.18	5.02	8.99	4.02
	R2T	61.77	13.1	15.63	7.71	79.04	79.97	79.38	Not supported*	
	LS	84.06	27.99	20.19	14.97	Not supported				

* It is a future work of [7].

Proof. Algorithm 4 is suitable to answer a star-join workload queries indirectly, by first perturbing a set of intermediate predicates under differential privacy via Algorithm 2, and then combining their predicates to answer the star-join workload queries. Thus, Algorithm 4 satisfies ϵ-differential privacy. □

Besides the privacy guarantee, we now conduct theoretical study over the utility of the Predicate Mechanism.

Theorem 5.6 (Loose Bound of Predicate Mechanism). *Let Q be the star-join query with n dimension tables and a fact table, the variance of using Predicate Mechanism is $(\frac{2n^2}{\epsilon^2})^n \cdot \prod_{i=1}^{n} dom(a_i)^2$.*

Proof. Since the Predicate of star-join query, *i.e.,* Φ, is the conjunction of each predicate of dimension tables ϕ_{a_i}, $\Phi = \prod_{i=1}^{n} \phi_{a_i}$, and each predicate $\hat{\phi}_{a_i}$ satisfies $\frac{\epsilon}{n}$-differential privacy, the variance of each predicates $\hat{\phi}_{a_i}$ is $2(\frac{n \cdot dom(a_i)}{\epsilon})^2$ and the expectation is 0 due to the Laplace noise. In addition, as each dimension table is independent of each other, the variance of predicate mechanism is the multiplication of the variance of $\hat{\phi}_{a_i}$, $(\frac{2n^2}{\epsilon^2})^n \cdot \prod_{i=1}^{n} dom(a_i)^2$. □

Theorem 5.7 (Tight Bound of Predicate Mechanism). *Let Q be the star-join query with n dimension tables and a fact table, the variance of using Predicate Mechanism is $(\frac{2n^2}{\epsilon^2}) \cdot \sum_{i=1}^{n} dom(a_i)^2$.*

Proof. Since the input of Φ is the conjunction of the binary, then Φ can be expressed as an indicate function, $\Phi = \mathbb{I}[\sum_{i=1}^{n} \phi_{a_i} = n]$. Moreover, each predicate $\hat{\phi}_{a_i}$ satisfies $\frac{\epsilon}{n}$-differential privacy, the variance of each predicates $\hat{\phi}_{a_i}$ is $2(\frac{n \cdot dom(a_i)}{\epsilon})^2$ and the expectation is 0 due to the Laplace noise. In addition, the dimension table is independent of each other and the introduction of an indication function does not cause any extra errors. Therefore, the variance of predicate mechanism is the sum of the variance of $\hat{\phi}_{a_i}$, $(\frac{2n^2}{\epsilon^2}) \cdot \sum_{i=1}^{n} dom(a_i)^2$. □

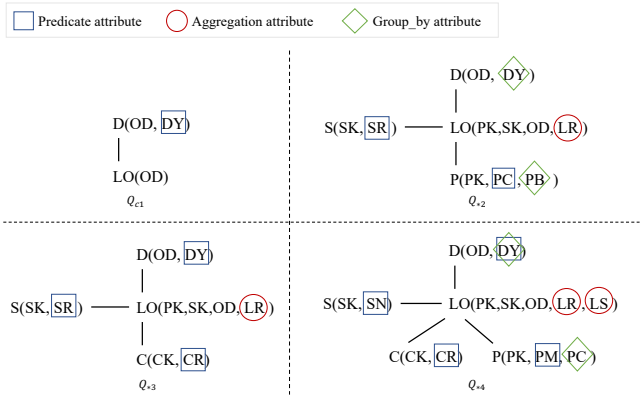

Figure 3: The structure of SSB queries

6 EXPERIMENTS

We conducted empirical study to test the performance for our model on a pair of benchmarking datasets. To evaluate the performance in various applications, the empirical study involves not only general star-join queries in OLAP scenarios, but also k-star counting queries in graph, which is a fundamental task in graph database and representative instance of star-join in specific applications.

For general star-join queries, we compare with a pair of state-of-the-art DP-compliant query schemes, namely R2T [7] and the local sensitivity-based mechanism (LS) [35]. For k-star counting queries, in line with [7], we compare PM with R2T and naive truncation with smooth sensitivity (TM) [16], which dominates LS in DP-compliant k-star tasks.

Dataset. To test the performance on the general star-join queries, we perform experiments using the Star Schema Benchmark (SSB) [27], a variation of the TPC-H benchmark widely adopted in star-join studies [26, 33]. It changes the snowflake model adopted in TPC-H

Table 2: Comparison between PM, R2T, TM on k-star queries by varying ϵ.

Privacy budget			$\epsilon = 0.1$		$\epsilon = 0.5$		$\epsilon = 1$	
Result type			Relative error(%)	Time(s)	Relative error(%)	Time(s)	Relative error(%)	Time(s)
Deezer	Q_{2*}	PM	38.25	0.14	35.91	0.11	30.53	0.11
		R2T	52.45	15.02	74.56	15.03	63.36	15.46
		TM	2431.4	5.53	339.55	5.27	279.18	4.9
	Q_{3*}	PM	65.06	0.84	58.85	1.25	56.67	1.15
		R2T	Over time limit					
		TM	385.75	164.05	306.49	164.45	117.3	160.77
Amazon	Q_{2*}	PM	17.67	0.67	11.41	0.60	7.39	0.75
		R2T	23.91	127.25	10.63	131.86	8.38	145.39
		TM	3750.34	80.4	482.01	83.51	42.03	76.33
	Q_{3*}	PM	16.25	4.62	14.78	4.70	7.90	4.33
		R2T	Over time limit					
		TM	Over time limit					

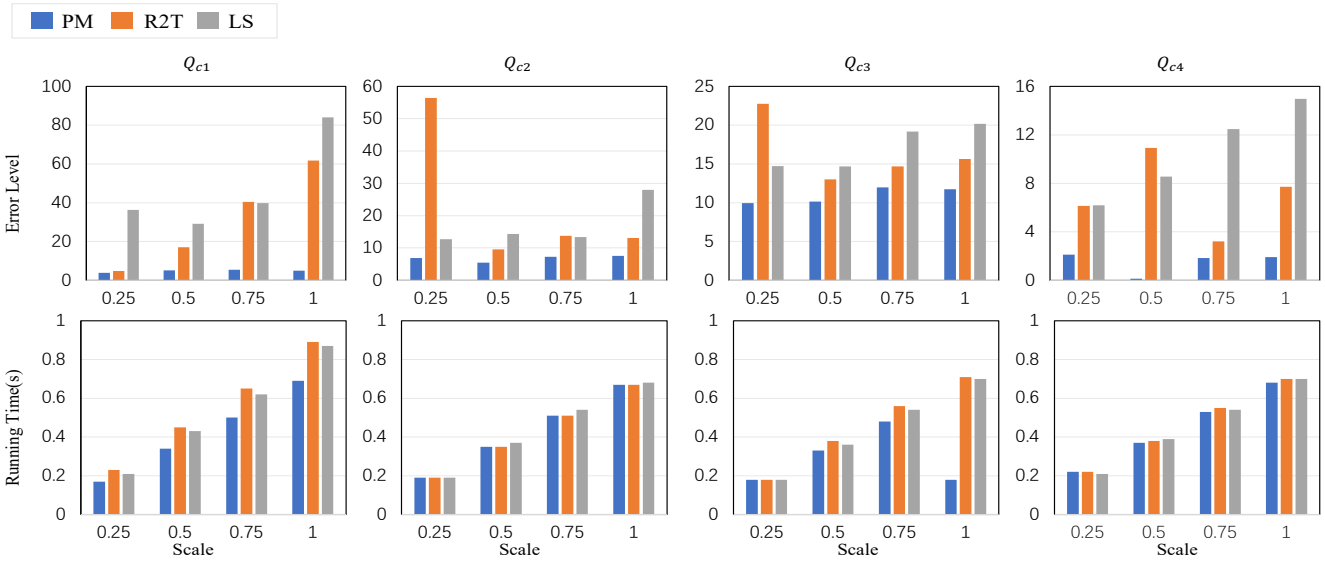

Figure 4: Running times and error level of PM, R2T, LS for different data scales (COUNT).

into a star model. SSB has a fact table and four dimension ones. Each dimension table contains hierarchical attributes, the value of which can be categorized into three types based on the hierarchy: large, medium, and small. For example, the *Customer* table contains attributes with different domain values of *city*, *region*, and *address*. In a star join query, the predicate only involves one of them.

For k-star counting queries, we adopt two real-world network datasets [19], namely **Deezer** and **Amazon**. The former collects all friendships of users from music streaming service Deezer, which contains friendship networks of users from 3 European countries and consists of 144000 nodes (*i.e.,* users) and 847000 edges (*i.e.,* friendships). The latter is an Amazon co-purchasing network, which contains 335000 nodes and 926000 edges. The k-star counting queries predicate refers to its node range, so the domain size is its number of vertices.

6.1 Setup

Queries. We tested 9 queries out of three standard star-join tasks from SSB, including counting queries $\{Q_{c1}, Q_{c2}, Q_{c3}, Q_{c4}\}$, sum queries $\{Q_{s2}, Q_{s3}, Q_{s4}\}$, and group-by queries $\{Q_{g2}, Q_{g4}\}$. As an example, $\{Q_{c1}\}$ involves a dimension table, $\{Q_{c2}, Q_{c3}\}$ contains 3 dimension tables, and Q_{c4} involves all the dimension tables. The structure of these queries are outlined in Figure 3.

For star-join workload queries, we use two types from the counting queries, $\{W_1, W_2\}$, in which W_1 contains all point constraints for one of three dimension tables and W_2 contains constraints for three dimension tables, one of which is a cumulative distribution (*i.e.,* each query sums the unit counts in a range $[1, i]$, where i is in the domain of an attribute). The $\{W_1, W_2\}$ as follows:

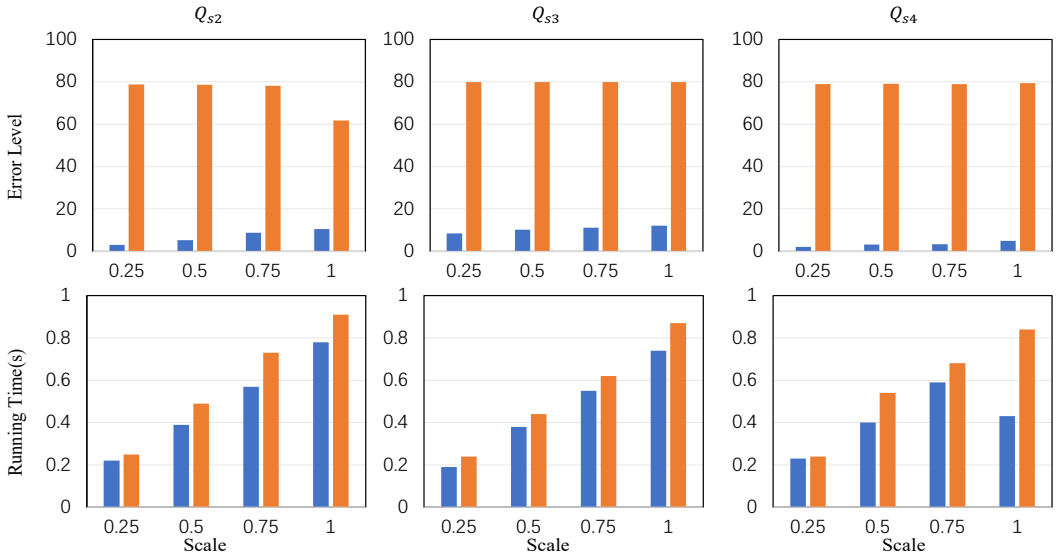

Figure 5: Running times and error level of PM and R2T for different data scales (SUM).

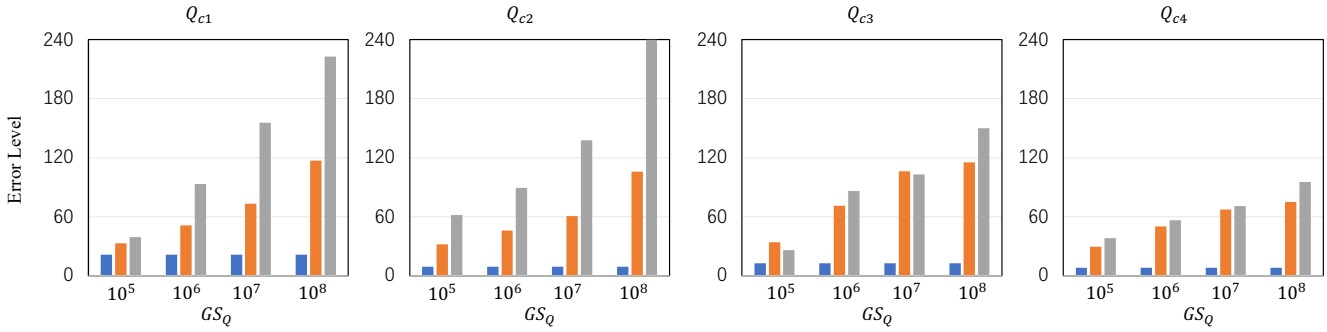

Figure 6: Error level of PM, R2T, LS for different GS_Q.

$$W_1 = \begin{bmatrix} 1000000 & 10000 & 10000 \\ 0100000 & 10000 & 10000 \\ 0010000 & 10000 & 10000 \\ 0001000 & 10000 & 10000 \\ 0000100 & 10000 & 10000 \\ 0000010 & 10000 & 10000 \\ 0000001 & 10000 & 01000 \\ 0011000 & 01000 & 01000 \\ 0001100 & 00100 & 01000 \\ 0000110 & 00010 & 01000 \\ 0000011 & 00001 & 01000 \end{bmatrix}, W_2 = \begin{bmatrix} 1000000 & 00100 & 10000 \\ 1100000 & 00100 & 10000 \\ 1110000 & 10000 & 10000 \\ 1111000 & 00100 & 01000 \\ 1111100 & 00010 & 00100 \\ 1111110 & 00001 & 10000 \\ 1111111 & 00100 & 01000 \end{bmatrix}.$$

For k-star counting queries, we test two different tasks: 2-star counting Q_{2*} and 3-star counting Q_{3*}.

Evaluation Metrics. Relative error is used as the utility measure and privacy budget is varied from {0.1, 0.2, 0.5, 0.8, 1}. In addition, we also evaluate the running time for all the compared solutions.

6.2 Empirical results

In each experiment we report the average response time of 10 independent runs, each of which is kept within a time limit (*i.e.,* 3 hours).

Utility. We tested the utility of different solutions by varying ϵ, and the result are shown in Table 1 and 2, respectively. As the privacy budget increases, error level gradually decreases as expected. In particular, according to Table 1, both PM and R2T achieve high utility under star-join count queries, while LS achieves bad utility except for very large ϵ. R2T achieves similar utility as PM on counting queries, but is much worse on sum queries. In all, Table 1 shows that PM achieves order-of-magnitude improvements over R2T and LS in terms of utility. More importantly, PM supports a variety of star-join queries comparing with R2T and LS. Remarkably, in all the star-join queries, PM consistently achieves an error below 20% (even $< 15\%$ when $\epsilon \geq 0.5$).

In addition, if comparing the performance between SSB and Deezer, PM performs better in the former. This is because the error of PM is proportional to the sum of domains according to our theoretical study in Section 5.4. Therefore, larger dimension tables in star-join queries lead to smaller relative errors. Compared with R2T and LS, PM exhibits much little change by varying ϵ. In general, the DP-starJ is more stable and accurate than R2T and LS in light of star-join queries. Simiarly, Table 2 also justifies the superiority

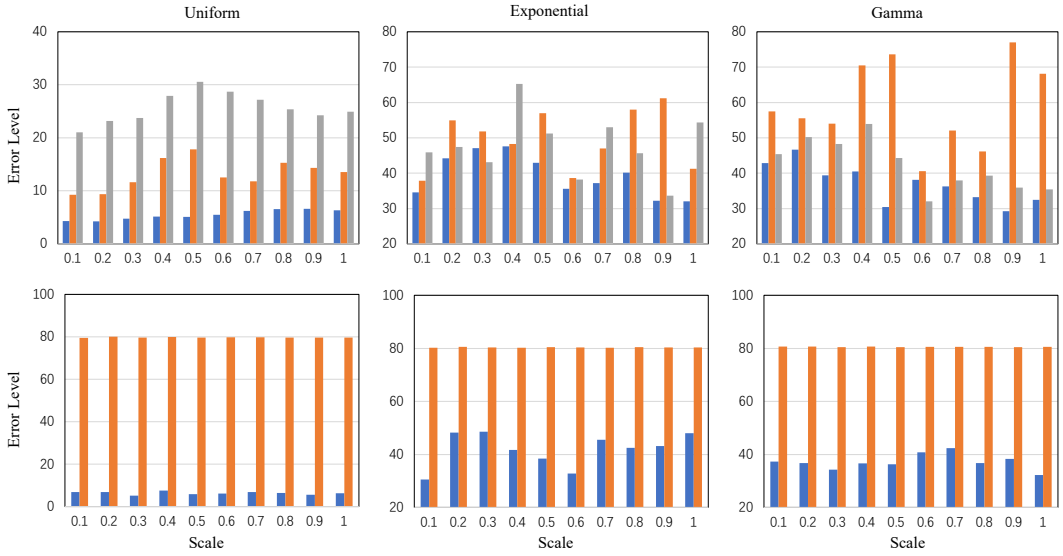

Figure 7: Error level of PM, R2T, LS for different distributions on Q_{c3}(top) and Q_{s3}(bottom) with different data scales.

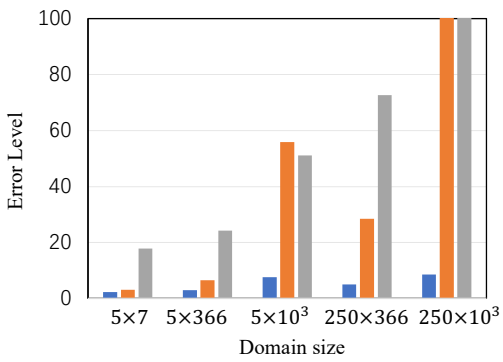

Figure 8: Error level of PM, R2T, LS for different domain sizes.

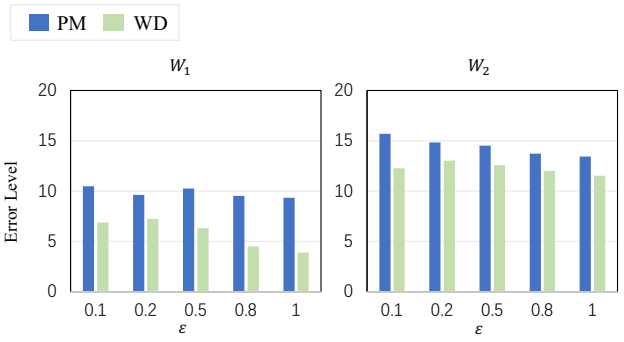

Figure 9: Error level of PM and WD for different ϵ.

of PM in terms of utility, offering an order-of-magnitude improvements over other methods in different cases. In workload queries, the error level of PM and WD mechanisms are shown in Figure 9. As the figure demonstrates, WD always introduces lower error than PM, especially on W_1.

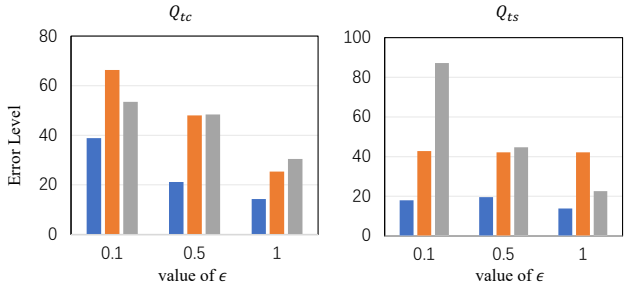

Figure 10: Error levels of various mechanisms on TPC-H queries by varying ϵ.

Efficiency. The running time of all mechanisms over the k-star counting queries are shown in Table 2. On **Deezer** dataset, across all the compared solutions, R2T can only complete within the 3-hour time limit on 3-star queries, although it achieves smaller errors on 2-star queries than TM. Comparing with both R2T and TM, PM is much faster (at least 40 times faster than TM) as it does not require additional data truncation.Both R2T and TM exceed the time limit on **Amazon** dataset, which can be attributed to an increase in the number of joins. R2T needs to solve linear programming problems to determine truncation thresholds, and TM involves local sensitivity computation, which leads to extra computational overhead. On the other hand, as the running time on SSB do not vary much either across approaches or under different privacy budgets, which have been shown in Figure 4 and 5, we select not to report it in Table 1 explicitly.

Scalability. In addition, we also test the scalability of the approaches by varying the volume of the database, using SSB with scale factors ranging from 0.25 to 1. The results are shown in Figure 4 and 5. Obviously, the error of PM barely increases with the data size. The reason is that our error only depends on domain size of attributes in queries, which does not change much by the scale of SSB data. On the other hand, the behavior of R2T is more

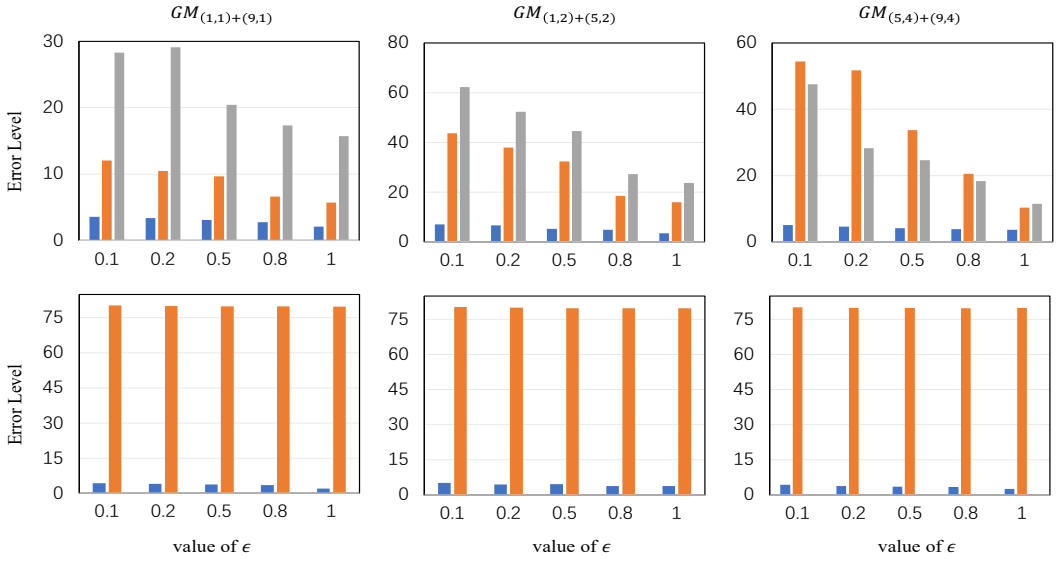

Figure 11: Error level of PM, R2T, LS for mixture of Gaussian distribution on Q_{c3}(top) and Q_{s3}(bottom) with different skewed parameter.

complicated. For Q_{c2} and Q_{c3}, its error decreases first but then increases later; for Q_{c4}, its error increases first but then decreases later. The reason is that R2T needs to choose the optimal result based on the truncation threshold, which is closely related to the scale of database instance. The utility of LS linearly increase with the data size as expected. In terms of running time, all mechanisms linearly increase with the data size, among which the increment of PM is smaller. Compared with the basic solutions, DP-starJ show superior performance in various star-join query types.

The impact of Domain size. To further evaluate the impact of domain size on PM, we extended the star-join count query on the SSB dataset, set up five queries with different domain value combinations involving two dimension table. The results are shown in Figure 8. Due to the increase in noise with the domain size, the error of the PM will experience a slight increase. When PM perturbs the predicate, its perturbation result is still within the domain value range, which weakens the impact of noise on the results to a certain extent. In addition, the error of PM is still orders of magnitude smaller than R2T and LS.

Different distributions. As shown in Figures 4 and 5, there are differences in the performance of PM in count and sum queries. In order to further investigate the reasons, we constructed data instances following different distributions on the SSB dataset. The results are shown in Figure 7. Firstly, the PM performs best on Uniform distribution, and the error gradually increases as the data distribution becomes skewed (*e.g.*,, Exponential and Gamma distributions). Secondly, for count queries, the error growth rate is higher. Lastly, with increasing data volume, the error of the PM decreases after an initial increase for summation queries. The main reason for this difference is that the result of summation queries depends on the values of the data itself, while the result of count queries depends on the data distribution. To further justify the impact of skewed data on the PM, extensive experiments are conducted by using data following a mixture of Gaussian distribution with different parameters. The results are shown in Figure 11. It is obvious that PM has a greater impact on count queries on skewed data. This observation partially suggests that count query results are more dependent on the data distribution.

Dependency on GS_Q. Our last set of experiments examine the effect GS_Q brings to the utilities of PM, R2T and LS. We conduct experiments using counting queries with different values GS_Q. The results are shown in Figure 6. It is observed that PM is insensitive with GS_Q as GS_Q of PM is only related to the queries. When GS_Q increases, the errors of R2T and LS increases rapidly.

Evaluation on snowflake query. To illustrate the effect of PM on snowflake queries, we select 2 queries from the TPC-H benchmark, referred to as Q_{tc} and Q_{ts}, which are count and sum queries, respectively. The results are shown in Figure 10, it is observed that PM outperforms both R2T and LS.

7 CONCLUSIONS

In this paper we present a novel solution to answer star-join query under differential privacy. We propose the definitions of neighboring database instances in different cases of star-join, taking into account the non-trivial number of foreign key constraints. Inspired by a latest output mechanism framework, we propose DP-starJ under DP for answering star-join queries, in which we design a new mechanism using predicate perturbation to achieve reasonable utility, efficiency and scalability.

REFERENCES

[1] Myrto Arapinis, Diego Figueira, and Marco Gaboardi. 2016. Sensitivity of counting queries. In *International Colloquium on Automata, Languages, and Programming (ICALP)*.

[2] Boaz Barak, Kamalika Chaudhuri, Cynthia Dwork, Satyen Kale, Frank McSherry, and Kunal Talwar. 2007. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 273–282.

[3] Jaroslaw Błasiok, Mark Bun, Aleksandar Nikolov, and Thomas Steinke. 2019. Towards instance-optimal private query release. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2480–2497.
[4] Kuntai Cai, Xiaokui Xiao, and Graham Cormode. 2023. PrivLava: Synthesizing Relational Data with Foreign Keys under Differential Privacy. *arXiv preprint arXiv:2304.04545* (2023).
[5] Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, Entong Shen, and Ting Yu. 2012. Differentially private spatial decompositions. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 20–31.
[6] Wei-Yen Day, Ninghui Li, and Min Lyu. 2016. Publishing graph degree distribution with node differential privacy. In *Proceedings of the 2016 International Conference on Management of Data*. 123–138.
[7] Wei Dong, Juanru Fang, Ke Yi, Yuchao Tao, and Ashwin Machanavajjhala. 2022. R2t: Instance-optimal truncation for differentially private query evaluation with foreign keys. In *Proceedings of the 2022 International Conference on Management of Data*. 759–772.
[8] Wei Dong and Ke Yi. 2021. Residual Sensitivity for Differentially Private Multi-Way Joins. In *Proceedings of the 2021 International Conference on Management of Data*. 432–444.
[9] Wei Dong and Ke Yi. 2022. A Nearly Instance-optimal Differentially Private Mechanism for Conjunctive Queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 213–225.
[10] Cynthia Dwork. 2006. Differential privacy. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33*. Springer, 1–12.
[11] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam 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 3*. Springer, 265–284.
[12] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.
[13] Cesar A Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing star join queries for data warehousing in microsoft sql server. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 1190–1199.
[14] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A simple and practical algorithm for differentially private data release. *Advances in neural information processing systems* 25 (2012).
[15] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.
[16] Shiva Prasad Kasiviswanathan, Kobbi Nissim, Sofya Raskhodnikova, and Adam D Smith. 2013. Analyzing Graphs with Node Differential Privacy.. In *TCC*, Vol. 13. Springer, 457–476.
[17] Fumiyuki Kato, Tsubasa Takahashi, Shun Takagi, Yang Cao, Seng Pei Liew, and Masatoshi Yoshikawa. 2022. HDPView: Differentially Private Materialized View for Exploring High Dimensional Relational Data. *arXiv preprint arXiv:2203.06791* (2022).
[18] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. Privatesql: a differentially private sql query engine. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1371–1384.
[19] Jure Leskovec and Andrej Krevl. 2016. SNAP datasets: Stanford large network dataset collection(2014). http://snap.stanford.edu/data
[20] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. 2015. The matrix mechanism: optimizing linear counting queries under differential privacy. *The VLDB journal* 24 (2015), 757–781.
[21] Seng Pei Liew, Tsubasa Takahashi, Shun Takagi, Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. 2022. Network shuffling: Privacy amplification via random walks. In *Proceedings of the 2022 International Conference on Management of Data*. 773–787.
[22] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 19–30.
[23] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially private join queries over distributed databases. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 149–162.
[24] Aleksandar Nikolov, Kunal Talwar, and Li Zhang. 2013. The geometry of differential privacy: the sparse and approximate cases. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 351–360.
[25] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2007. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 75–84.
[26] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 237–252.
[27] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.
[28] Catuscia Palamidessi and Marco Stronati. 2012. Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. *arXiv preprint arXiv:1207.0872* (2012).
[29] Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2012. Calibrating data to sensitivity in private data analysis. *arXiv preprint arXiv:1203.3453* (2012).
[30] Wahbeh Qardaji, Weining Yang, and Ninghui Li. 2013. Understanding hierarchical methods for differentially private histograms. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1954–1965.
[31] Wahbeh Qardaji, Weining Yang, and Ninghui Li. 2014. Priview: practical differentially private release of marginal contingency tables. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1435–1446.
[32] Uwe Röhm, Klemens Böhm, and Hans-Jörg Schek. 2000. OLAP query routing and physical design in a database cluster. In *Advances in Database Technology—EDBT 2000: 7th International Conference on Extending Database Technology Konstanz, Germany, March 27–31, 2000 Proceedings 7*. Springer, 254–268.
[33] Jimi Sanchez. 2016. A review of star schema benchmark. *arXiv preprint arXiv:1606.00295* (2016).
[34] Shun Takagi, Tsubasa Takahashi, Yang Cao, and Masatoshi Yoshikawa. 2021. P3GM: Private high-dimensional data release via privacy preserving phased generative model. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 169–180.
[35] Yuchao Tao, Xi He, Ashwin Machanavajjhala, and Sudeepa Roy. 2020. Computing local sensitivities of counting queries with joins. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 479–494.
[36] Tianhao Wang, Milan Lopuhaä-Zwakenberg, Zitao Li, Boris Skoric, and Ninghui Li. 2019. Locally differentially private frequency estimation with consistency. *arXiv preprint arXiv:1905.08320* (2019).
[37] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially private SQL with bounded user contribution. *Proceedings on privacy enhancing technologies* 2020, 2 (2020), 230–250.
[38] Xiaokui Xiao, Guozhang Wang, and Johannes Gehrke. 2010. Differential privacy via wavelet transforms. *IEEE Transactions on knowledge and data engineering* 23, 8 (2010), 1200–1214.
[39] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, Ge Yu, and Marianne Winslett. 2013. Differentially private histogram publication. *The VLDB journal* 22 (2013), 797–822.
[40] Ganzhao Yuan, Zhenjie Zhang, Marianne Winslett, Xiaokui Xiao, Yin Yang, and Zhifeng Hao. 2015. Optimizing batch linear queries under exact and approximate differential privacy. *ACM Transactions on Database Systems (TODS)* 40, 2 (2015), 1–47.
[41] Sepanta Zeighami, Ritesh Ahuja, Gabriel Ghinita, and Cyrus Shahabi. 2021. A neural database for differentially private spatial range queries. *arXiv preprint arXiv:2108.01496* (2021).
[42] Xiaojian Zhang, Rui Chen, Jianliang Xu, Xiaofeng Meng, and Yingtao Xie. 2014. Towards accurate histogram publication under differential privacy. In *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 587–595.
[43] Shuyuan Zheng, Yang Cao, and Masatoshi Yoshikawa. 2022. Secure Shapley Value for Cross-Silo Federated Learning. *arXiv preprint arXiv:2209.04856* (2022).

A LIST OF QUERIES AND THEIR DOMAIN SIZES

We provided the detailed queires, predicates and their corresponding domain sizes on SSB queries and k-star queries in this section.

The SSB queries as follows:

Q_{c1} : 7. The domain size of predicate $Date.year$ is 7.

```
SELECT count(*) FROM Date, Lineorder
WHERE Lineorder.orderdate = Date.DK
  AND Date.year = 1993;
```

Q_{c2} : 25×5, which means that the domain sizes of predicates $Part.category$ and $Supplier.region$ are 25 and 5, respectively.

```
SELECT count(*)
FROM Date, Lineorder, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineorder.orderdate = Date.DK
```

```
  AND Part.category = 'MFGR#12'
  AND Supplier.region = 'AMERICA';
```

Q_{c3} : $5 \times 5 \times 7$.

```
SELECT count(*)
FROM Date, Lineorder, Customer, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.CK = Customer.CK
  AND Lineorder.orderdate = Date.DK
  AND Customer.region = 'ASIA'
  AND Supplier.region = 'ASIA'
  AND Date.year between 1992 and 1997;
```

Q_{c4} : $5 \times 25 \times 7 \times 5$.

```
SELECT count(*)
FROM Date, Lineorder, Customer, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineroder.CK = Customer.CK
  AND Lineorder.orderdate = Date.DK
  AND Customer.region = 'AMERICA'
  AND Supplier.nation = 'UNITED STATES'
  AND Date.year between 1997 and 1998
  AND Part.mfgr = 'MFGR#1'
  OR Part.mfgr = 'MFGR#2';
```

Q_{s2} : 25×5.

```
SELECT sum(Lineorder.revenue)
FROM Date, Lineorder, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineorder.orderdate = Date.DK
  AND Part.category = 'MFGR#12'
  AND Supplier.region = 'AMERICA';
```

Q_{s3} : $5 \times 5 \times 7$.

```
SELECT sum(Lineorder.revenue)
FROM Date, Lineorder, Customer, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.CK = Customer.CK
  AND Lineorder.orderdate = Date.DK
  AND Customer.region = 'ASIA'
  AND Supplier.region = 'ASIA'
  AND Date.year between 1992 and 1997;
```

Q_{s4} : $5 \times 25 \times 7 \times 5$.

```
SELECT sum(Lineorder.revenue)
FROM Date, Lineorder, Customer, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineroder.CK = Customer.CK
  AND Lineorder.orderdate = Date.DK
  AND Customer.region = 'AMERICA'
  AND Supplier.nation = 'UNITED STATES'
  AND Date.year between 1997 and 1998
  AND Part.mfgr = 'MFGR#1'
  OR Part.mfgr = 'MFGR#2';
```

Q_{g2} : 25×5.

```
SELECT sum(Lineorder.revenue), Date.year, Part.brand
FROM Date, Lineorder, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineorder.orderdate = Date.DK
  AND Part.category = 'MFGR#12'
  AND Supplier.region = 'AMERICA'
Group by Date.year, Part.brand
Order by Date.year, Part.brand;
```

Q_{g4} : $5 \times 25 \times 7 \times 5$.

```
SELECT sum(Lineorder.revenue - Lineorder.supplycost),
  Date.year, Part.category
FROM Date, Lineorder, Customer, Part, Supplier
WHERE Lineroder.SK = Supplier.SK
  AND Lineroder.PK = Part.PK
  AND Lineroder.CK = Customer.CK
  AND Lineorder.orderdate = Date.DK
  AND Customer.region = 'AMERICA'
  AND Supplier.nation = 'UNITED STATES'
  AND Date.year between 1997 and 1998
  AND Part.mfgr = 'MFGR#1' OR Part.mfgr = 'MFGR#2'
Group by Date.year, Part.category
Order by Date.year, Part.category;
```

The k-star queries on **Deezer** and **Amazon** datasets as follows: The k-star counting queries predicate refers to its node range, so the domain size is its number of vertices.

Deezer: the domain size of k-star queries is 144000.

Q_{2*} :

```
SELECT count(*)
FROM Edge AS R1, Edge AS R2, Edge AS R3
WHERE R1.from_id = R2.from_id
  AND R1.to_id < R2.to_id
  AND R1.from_id between 1 and 144000;
```

Q_{3*} :

```
SELECT count(*)
FROM Edge AS R1, Edge AS R2
WHERE R1.from_id = R2.from_id
  AND R1.from_id = R3.from_id
  AND R1.to_id < R2.to_id
  AND R2.to_id < R3.to_id
  AND R1.from_id between 1 and 144000
  AND R3.from_id between 1 and 144000;
```

Amazon: the domain size of k-star queries is 335000.

Q_{2*} :

```
SELECT count(*)
FROM Edge AS R1, Edge AS R2
WHERE R1.from_id = R2.from_id
  AND R1.to_id < R2.to_id
  AND R1.from_id between 1 and 335000;
```

Q_{3*} :

```
SELECT count(*)
FROM Edge AS R1, Edge AS R2, Edge AS R3
WHERE R1.from_id = R2.from_id
  AND R1.from_id = R3.from_id
  AND R1.to_id < R2.to_id
  AND R2.to_id < R3.to_id
  AND R1.from_id between 1 and 335000
  AND R3.from_id between 1 and 335000;
```