How Differential Privacy Accelerates Multi-Party Computation in Subgraph Matching

Shiyuan Tang
Beijing Institute of Technology
Beijing, China
shiyuantg@gmail.com

Zhikun Zhang
Zhejiang University
Hangzhou, China
zhikun@zju.edu.cn

Quan Yuan
Zhejiang University
Hangzhou, China
yq21@zju.edu.cn

Lianpeng Qiao
Beijing Institute of Technology
Beijing, China
qiaolp@bit.edu.cn

Ye Yuan
Beijing Institute of Technology
Beijing, China
yuan-ye@bit.edu.cn

Yunjun Gao
Zhejiang University
Hangzhou, China
gaoyj@zju.edu.cn

ABSTRACT

With the increasing adoption of outsourced graph analytics, privacy-preserving subgraph matching has become critical for protecting sensitive structural information. However, existing approaches fail to provide comprehensive protection against structural inference attacks by servers, require additional result verification with trust assumptions between the data owner and the data analyst, and generate massive redundant results due to query graph symmetries. To address these issues, we propose PRISM, a new approach that reformulates privacy-preserving subgraph matching as secure multi-round relational joins in a *multi-party computation* (MPC) setting. PRISM achieves information-theoretic privacy through secret sharing and enables the data analyst to directly obtain exact results without additional processing. A key challenge is that naively applying secure joins requires padding results with dummy values far exceeding actual sizes to maintain obliviousness, causing prohibitive overhead. We address this through two complementary optimizations. At the architectural level, we design a decomposition-matching-assembly strategy that exploits query topology through balanced star decomposition and topology reuse, significantly reducing required joins. At the algorithmic level, PRISM integrates *differential privacy* (DP) to release statistical synopses of the join key, accelerating operations through: (i) hierarchical histogram-based differentially private index enabling parallel execution via lossless partitioning, and (ii) hybrid upper bound estimation combining differentially private statistics-based bounds with the worst-case bounds to compress the join results. Experiments on real-world datasets show that PRISM successfully processes complex queries where state-of-the-art methods fail, and achieves 29.6× speedup and 24.9× memory reduction on simple queries.

PVLDB Reference Format:
Shiyuan Tang, Zhikun Zhang, Quan Yuan, Lianpeng Qiao, Ye Yuan, and Yunjun Gao. How Differential Privacy Accelerates Multi-Party Computation in Subgraph Matching. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tsy529/PRISM.

1 INTRODUCTION

With the rapid advancement of graph analytics, subgraph matching has emerged as a fundamental operation widely used in social network analysis [17], financial fraud detection [31], biological network analysis [32], *etc.* Traditional local query methods are often limited by computational resources and storage capabilities [36, 53]. To address these limitations, the outsourced computation paradigm has been increasingly adopted [1], where the data owner delegates their graph data to powerful cloud service providers for query processing without maintaining expensive local infrastructure. However, this introduces critical privacy concerns: graph data often contains sensitive structural information, as even partial leakage can expose sensitive relationships and patterns [13]. Therefore, privacy-preserving subgraph matching in outsourced scenarios [11, 12, 18, 49] has emerged to enable efficient query processing while protecting sensitive information throughout the computation process.

Existing Solutions. Previous privacy-preserving subgraph matching approaches under outsourcing typically follow a three-phase workflow. In the initial setup phase, the *data owner* encrypts or obfuscates the sensitive data and uploads it to the *server*. During the query processing phase, when a *data analyst* submits a query request, the server processes it over the protected graph to generate candidate results. In the final verification phase, these candidates are returned to the data analyst for plaintext filtering against the original graph to obtain exact results. To enable this workflow while preserving privacy, current research primarily focuses on designing candidate generation techniques under different privacy-preserving strategies, which can be classified into three categories based on their protection mechanisms.

Concretely, k-automorphism-based methods [12, 18, 21, 43, 44] transform the data graph by partitioning vertices into k blocks and introducing noisy edges to ensure structural anonymization, where the server returns candidates containing false positives from noisy edges. Structure-based encryption methods [48, 49] precompute and encrypt subgraph structures with well-defined boundaries (*e.g.*, balls), enabling the server to return encrypted candidate structures through encrypted operations. Feature-based indexing

methods [11] construct encrypted indices from mined frequent subgraph features, allowing the server to identify candidate graphs through encrypted feature matching. However, these methods still suffer from the following common limitations.

• *Inadequate structural protection.* Existing methods fail to provide sufficient protection for the graph topology against the server. They either anonymize only node-level structure while leaving edge patterns and degree distributions exposed, or require the server to compute and return candidate subgraphs that inevitably contain structural information beyond actual query results. This structural leakage enables potential inference attacks that can compromise sensitive relationships and patterns in the graph [25].

• *Additional processing requirements.* All existing approaches require the server to return candidates that need further processing to obtain exact results. This introduces additional computational overhead for both the data owner and the data analyst for plaintext verification and filtering. Furthermore, they require the data owner to share the original graph G with the data analyst for verification [21], which represents an additional trust consideration that limits their applicability in scenarios involving the semi-trusted or untrusted data analyst.

• *Symmetry-induced redundancy.* When query graphs contain symmetric vertices, existing methods enumerate all permutations of symmetric mappings, resulting in significant redundancy. For example, a triangle query generates $3! = 6$ redundant copies of each match, while a k-clique query produces $k!$ duplicates. This redundancy exponentially inflates both result size and computational cost, severely limiting practical scalability.

Our Proposal. To address these issues, we propose PRISM, a new approach that fundamentally advances privacy-preserving subgraph matching in outsourced scenarios. Unlike existing approaches that rely on owner-analyst trust or sacrifice structural privacy, PRISM enables the untrusted data analyst to obtain exact results directly from servers, while providing comprehensive structural privacy protection throughout the entire query process.

• *Strong structural privacy.* We reformulate privacy-preserving subgraph matching under outsourcing as secure join queries in a *multi-party computation* (MPC) framework, where the data graph's structural information is encoded into relational tables, which are then secret-shared and distributed among three noncolluding servers. All query processing is performed entirely on secret shares through MPC protocols, with the servers returning only secret shares of results to the data analyst for independent reconstruction. This design achieves information-theoretic privacy for graph structure, resolving the limited structural protection in previous studies.

• *Efficient query design.* We introduce a decomposition-matching-assembly architecture that exploits the topology of the query graph to minimize expensive secure join operations, which is the primary performance bottleneck. In the decomposition phase, we apply balance processing to partition the query graph into stars while minimizing the maximum star size. In the matching phase, we introduce a topology-reuse mechanism that reuses the biggest star's matching intermediate results for all smaller stars, exploiting the unique property of stars that size uniquely determines topology. This ensures that only the biggest star requires explicit matching against the data graph. In the assembly phase, the matching results are combined through secure joins in a greedy order that minimizes the size of intermediate results.

• *Accurate result guarantee.* We propose two secure constraint verification protocols to validate the join results. Injectivity verification ensures that the results obtained by the data analyst strictly satisfying subgraph matching constraints, while symmetry verification eliminates redundant symmetric results caused by query graph automorphisms, all independent of the data owner filtering. This eliminates the need for owner-analyst trust assumptions in the query process, enabling the data analyst to obtain accurate results directly from the servers.

DP-Accelerated MPC. While the above techniques provide strong privacy and correctness guarantees, multi-round secure joins still impose prohibitive overhead due to the strict security requirements of MPC. Concretely, statistical information about tables can substantially improve join efficiency, but such information typically contains sensitive patterns and cannot be directly published. Therefore, we leverage *differential privacy* (DP) to release privacy-preserving statistical synopses of the join key, including the differentially private index (DPIdx) and the differentially private maximum frequency (DPMF), accelerating secure join execution from two perspectives under provable privacy guarantees.

• *Compressed join results.* MPC protocols must pad dummy values to ensure obliviousness, resulting in the join results far exceeding their actual sizes. We integrate two complementary strategies for tighter upper bound estimation: the MF bound [20] based on DPMF for statistics-based estimation, and the AGM bound (proposed by Atserias, Grohe, and Marx) [5] for the worst-case estimation. This hybrid approach dramatically compresses the join results while preserving obliviousness under DP guarantees.

• *Parallel join execution.* We propose building DPIdx based on differentially private hierarchical histograms to enable parallel secure join execution. DPIdx partitions data into buckets based on join keys while ensuring lossless coverage. When both tables are indexed on their join keys, secure joins can be executed in parallel between matching buckets, significantly improving efficiency.

Evaluation. We conduct extensive experiments with various query patterns on three real-world graph datasets to demonstrate the superiority of PRISM. Since no prior work addresses privacy-preserving subgraph matching under MPC, we implement baselines by adapting both traditional MPC join protocols and state-of-the-art MPC query processing techniques to this problem. The experimental results show that the baselines can only handle simple queries, while PRISM efficiently processes complex queries. Even for simple queries, the baselines incur 29.6× higher time overhead and 24.9× higher memory overhead compared to PRISM, with the performance gap widening as the query complexity increases. We further conduct ablation studies on optimization techniques in PRISM and validate its scalability.

2 PRELIMINARIES

2.1 Subgraph Matching

Subgraph matching is a fundamental graph query operation with wide applications that identifies instances of a query pattern in a larger graph [15].

Formally, let g be an undirected, unlabeled graph with vertices $V(g)$ and edges $E(g)$. For vertex $v \in V(g)$, define neighbors $N(v) = \{u|(u,v) \in E(g)\}$ and degree $d(v) = |N(v)|$. A subgraph satisfies $V(g') \subseteq V(g)$ and $E(g') \subseteq E(g)$.

Subgraph matching enumerates all subgraphs of a data graph G isomorphic to query Q. A *match* of Q in G is defined as a mapping $f : V(Q) \to V(G)$ satisfying the following conditions:

- **Injectivity:** For any pair of vertices $v_i, v_j \in V(Q)$, if $v_i \neq v_j$ then $f(v_i) \neq f(v_j)$.
- **Edge Consistency:** For every edge $(v_i, v_j) \in E(Q)$, the edge $(f(v_i), f(v_j))$ must exist in G.

Since each match f can be viewed as a tuple, all matches could form a relational table $R(Q) = (A, T)$ with attributes $A = V(Q)$ and tuples

$$T = ((f(v_1), f(v_2), \dots) \mid f \text{ is a match}). \tag{1}$$

Relational Join Approach. In computing subgraph matches, unlike backtracking-based enumeration that directly explores the search space, we adopt the relational join approach, which naturally preserves privacy by representing graph structures as relational tables amenable to secure computation.

Specifically, subgraph matching via relational joins decomposes query graphs into smaller structures (*i.e.,* join units), incrementally merging partial matches using multi-round two-way joins. A hierarchical join tree specifies the join order: leaves represent join units, internal nodes represent partial matches, and the root denotes the final result. If all internal nodes of the join tree have at least one join operation as their child, the tree is called a left-deep tree [22].

Symmetric Duplication. Computing matching results using joins alone may produce invalid (violating injectivity) or duplicate matches (due to symmetry). The former is straightforward to understand, so we now provide a further explanation of the latter. For the query Q, the symmetry partitions $V(Q)$ into several symmetric vertex groups, denoted as $\text{Svg}(Q) = \{C_1, \dots, C_k\}$. Each valid match repeats due to automorphisms, causing redundancy by a factor of $\prod_{i=1}^{k} |C_i|!$.

In this paper, we adopt the secure constraint verification protocol to address this issue. In this protocol, the injectivity check component ensures that each tuple represents a valid match, while the symmetry check component guarantees that there are no redundant matches in the matching results.

2.2 Secure Multi-Party Computation

Secure multi-party computation (MPC) enables multiple parties to jointly compute a function while keeping their private inputs confidential, except for the final result. Following prior works [6, 24, 29], we adopt a 3-party computation (3PC) setting with an honest majority (at most one corrupted party), which provides lower communication overhead. We consider three parties P_1, P_2, P_3, using the cyclic notation (*e.g.,* P_{i-1}, P_{i+1} relative to P_i).

Secret Sharing. A core technique in MPC is to split a secret into multiple shares, allowing only a specific number of parties to reconstruct the secret. In our presentation, encrypted data is represented by default using the (2,3)-replicated secret sharing scheme [2]. In this scheme, a secret x is shared as three random elements x_1, x_2, and x_3 that satisfy $x_1 \oplus x_2 \oplus x_3 = x$, and the party P_i holds shares (x_i, x_{i+1}). For brevity, we refer to shares of $[\![x]\!]$ as the tuple (x_1, x_2, x_3), where each party holds a pair of random elements as in replicated secret sharing. Reconstruction involves each P_i sending x_i to P_{i+1}, allowing local recovery $x = x_1 \oplus x_2 \oplus x_3$. This scheme tolerates one corrupted party.

The (2,3)-replicated secret sharing scheme enables efficient basic operations with substantially lower communication costs for Boolean AND operations between shared values compared to 2-PC using Beaver triples. Due to space constraints, we detail the three-party secure computation procedure using basic Boolean operations in Appendix A.

Note that secret sharing generalizes to vectors ($[\![V]\!]$), and relational tables ($[\![R]\!] = (A, [\![T]\!], [\![isNull]\!])$), where $[\![isNull]\!]$ indicates tuple validity (0 means valid).

Circuit Evaluation. Building upon secret sharing, the Goldreich-Micali-Wigderson (GMW) protocol [19] was first proposed to represent computation as Boolean circuits where each wire's value is expressed as secret-shared values. The MPC circuit evaluation protocol takes secret-shared inputs $[\![x]\!]$ and a circuit C as input, and outputs a secret-shared result $[\![y]\!] = C(x)$. The ideal functionality $\mathcal{F}_{\text{MPC}}$ ensures secure evaluation and can be instantiated via established MPC frameworks [28]. While circuit evaluation offers generality, it is often less efficient than protocols tailored to specific problems. In the following, we introduce two widely used protocols that will be employed throughout this paper.

Secure Join. Join operations combine related data from multiple tables based on common attributes. The secure join protocol takes shared relations $[\![R]\!]$ and $[\![S]\!]$ as input and outputs their join result $[\![R \bowtie S]\!]$. The ideal functionality $\mathcal{F}_{\text{Join}}$ securely matches tuples on shared join keys without revealing the original relations. It supports duplicate keys and can be implemented using existing approaches [6], achieving $O(n \log n)$ time complexity in $O(\log n)$ rounds.

Secure Sort. Sorting orders data according to specified attributes. The secure sort protocol takes a shared vector $[\![V]\!]$ or relation $[\![R]\!]$ with a designated sorting attribute $A_j \in R.A$, and outputs the sorted result. The ideal functionality $\mathcal{F}_{\text{Sort}}$ ensures the sorting process leaks no ordering information about the original data [4], requiring $O(n \log n)$ communication rounds.

2.3 Differential Privacy

Differential privacy (DP) is a rigorous mathematical framework that protects individual privacy when analyzing and releasing statistical data. The core idea is to introduce a controlled amount of random noise into the statistical result so that the inclusion or exclusion of any single record has a limited impact on the overall result. In its more general form, an algorithm is said to satisfy (ϵ, δ)-DP if for any two neighboring datasets that differ in a single entry, the probability of any specific outcome changes by at most a multiplicative factor of $\exp(\epsilon)$ plus an additive term δ. Formally, we have the following definition.

Definition 2.1 ((ϵ, δ)-Differential Privacy). An algorithm $\mathcal{A}$ satisfies (ϵ, δ)-differential privacy ((ϵ, δ)-DP), where $\epsilon \geq 0$ and $\delta \geq 0$, if and only if for any pair of neighboring datasets D and D', we have

$$\forall T \subseteq \mathit{Range}(\mathcal{A}) : \Pr[\mathcal{A}(D) \in T] \leq e^{\epsilon} \Pr[\mathcal{A}(D') \in T] + \delta, \tag{2}$$

where $Range(\mathcal{A})$ denotes the set of all possible outputs of the algorithm $\mathcal{A}$.

Laplace Mechanism. The Laplace mechanism is one of the most widely used mechanisms for DP by adding noise to the output of a numerical function. For a query function $f : \mathcal{D} \to \mathbb{R}$ and any neighboring datasets D and D', the global sensitivity is defined as $\Delta_f = \max_{D,D'} |f(D) - f(D')|$. The Laplace mechanism achieves ϵ-DP by computing $\mathcal{A}_f(D) = f(D) + \text{Lap}(\Delta_f/\epsilon)$, where the noise is drawn from the Laplace distribution with a probability density function $\Pr[\text{Lap}(\Delta_f/\epsilon) = x] = \frac{\epsilon}{2\Delta_f} \exp(-\frac{\epsilon|x|}{\Delta_f})$.

Properties of (ϵ, δ)-DP. When applying multiple DP algorithms, understanding how privacy loss accumulates or remains invariant under further processing is critical. The composition properties of DP are typically categorized as follows.

- **Sequential composition.** Consider the DP algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\}$, where each $\mathcal{A}_i$ satisfies (ϵ_i, δ_i)-DP. When applied sequentially in the same dataset, the composite algorithm satisfies $(\sum_{i=1}^{k} \epsilon_i, \sum_{i=1}^{k} \delta_i)$-DP.
- **Parallel composition.** If algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k\}$ operate on disjoint subsets of the data, where $\mathcal{A}_i$ satisfies (ϵ_i, δ_i)-DP, then the overall algorithm satisfies $(\max_{i\in[1,k]} \epsilon_i, \max_{i\in[1,k]} \delta_i)$-DP.
- **Post-processing.** Given a algorithm $\mathcal{A}$ satisfying (ϵ, δ)-DP, then for any function f, the composition $f(\mathcal{A})$ also satisfies (ϵ, δ)-DP. That is, no additional privacy loss is incurred by any further processing of $\mathcal{A}$'s output.

3 SYSTEM OVERVIEW

3.1 Problem Formulation

Application Scenarios. We consider the scenario consisting of three primary roles: data owner, data analyst, and three servers (as shown in Figure 1). The *data owner*, typically an enterprise or organization, possesses substantial sensitive data modeled as unlabeled, undirected graphs (note that while we focus on structural privacy and thus consider simplified graph models, our approach remains general and can be extended to labeled graphs), but has limited computational capabilities. The *data analyst* can be any individual user, another organization, or even the data owner itself, without local computational resources. All query requests must be authorized by the data owner, who has visibility into the request content and grants permission for query execution. The *servers* leverage their storage and computational capacities to execute secure protocols, delivering trustworthy and efficient query services without exposing sensitive graph structures.

Threat Model. We adopt an *honest-but-curious* adversary model following mainstream security assumptions in MPC [2, 45]. In our three-server setting, we assume an honest majority, meaning at most one server can be an honest-but-curious adversary who faithfully executes protocols but may attempt to infer sensitive information from observed data. We assume non-collusion among servers. The data owner is fully trusted, while the data analyst is also considered a potential adversary who should only obtain query results without learning any additional information about the data graph beyond their authorized queries.

Problem Definition. In this paper, we consider the following problem: *Given a data analyst holding a query graph Q and a data owner holding a sensitive data graph G, how do we achieve privacy-preserving subgraph matching of Q in G in the outsourcing scenarios where the servers follow the threat model described above?* Addressing the above problem requires achieving three objectives:

(1) **Comprehensive Structural Protection.** Comprehensively protect the structural information of the data graph against servers throughout the subgraph matching;
(2) **Accurate Matching Results.** Enable servers to return accurate results without requiring additional processing, thereby eliminating additional computational overhead and trust assumptions between the data owner and the data analyst;
(3) **Low Computational Overhead.** Achieve the above privacy and accuracy guarantees while maintaining practical efficiency.

3.2 Intuition

Existing solutions fail to meet our objectives. Concretely, solutions that rely on graph obfuscation or encryption of specific structures inevitably leak structural patterns to the servers during query processing, failing to achieve Objective (1). For Objective (2), returning candidates to the data analyst for verification requires collaboration between the data owner and the data analyst, which conflicts with our threat model, where the data analyst may be adversarial. Additionally, existing works do not consider result redundancy due to query graph symmetries.

To achieve comprehensive structural protection, we recognize that the join-based subgraph matching encodes graph structural information into relational tables. These tables can be secret-shared and processed using MPC protocols to execute join operations, perfectly concealing all sensitive information securely. Furthermore, MPC's strong programmability and composability naturally address the problem of inaccurate results.

However, naively applying MPC protocols incurs prohibitive overhead due to strict security requirements. To meet Objective (3), we conduct optimization at two complementary levels. At the architectural level, traditional join-based subgraph matching approaches are not designed to minimize the number of joins, yet the high cost of secure joins makes such optimization crucial. In this paper, we effectively exploit the query topology through balanced star decomposition and topology reuse, significantly reducing required joins. At the algorithmic level, statistical information about tables to be joined can significantly facilitate join operations, but directly using true statistics poses privacy risks. Therefore, we integrate DP to estimate the statistical synopses of the join key, ensuring structural privacy of the data graph while achieving excellent utility.

3.3 Overview

Since the statistical synopses in PRISM depend only on the data graph, they can be precomputed offline in the data preparation phase and reused across multiple queries to accelerate subsequent online queries. When the data analyst submits queries, the online query processing phase leverages these precomputed synopses for efficiency. We detail the PRISM architecture with two primary phases as shown in Figure 1.

Offline Data Preparation. As shown in the right part of Figure 1, the data owner first constructs edge matches $R(e)$ from the data graph G and secret-shares them among the servers. To support

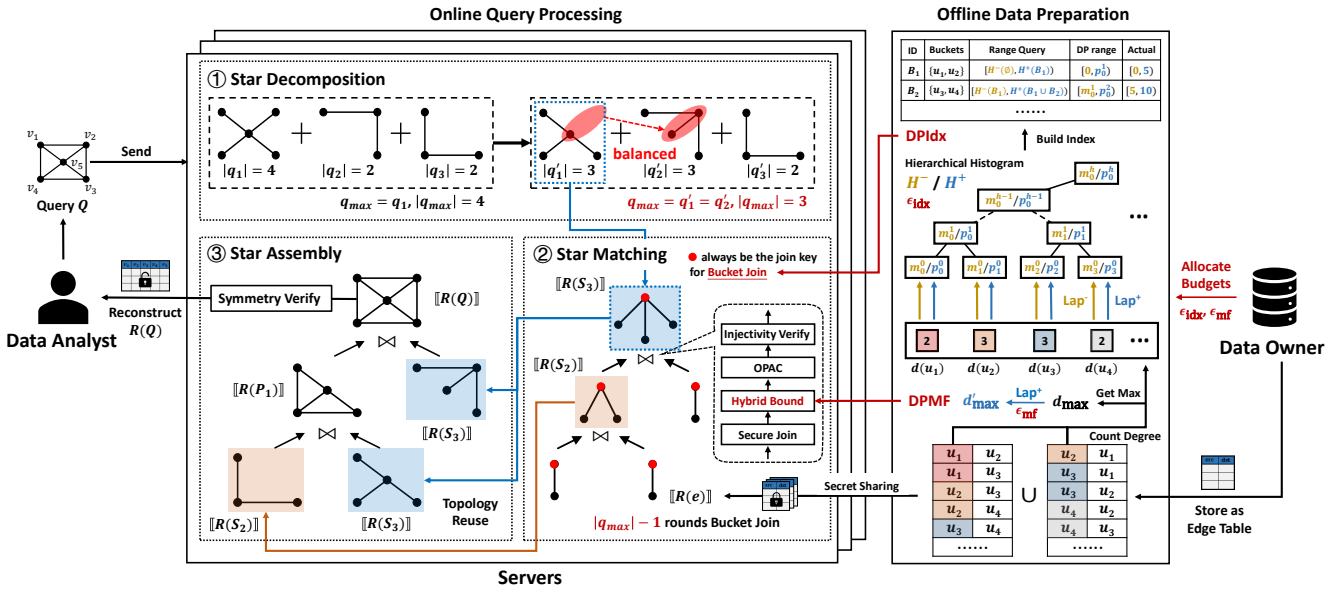

Figure 1: Overview of PRISM. PRISM **has two phases: data preparation and query processing. In the data preparation phase, the data owner concatenates the data graph's edge table with its transpose as edge matching result $R(e)$, then adds one-side Laplace noise ($\text{Lap}^+/\text{Lap}^-$) using budgets ϵ_{mf} and ϵ_{idx} to compute two synopses for any $R(e)$ column: (1) differentially private maximum frequency (DPMF) and (2) differentially private index (DPIdx) constructed via hierarchical histograms ($\mathcal{H}^+/\mathcal{H}^-$). These synopses support Hybrid Bound and Bucket Join in query phase. In the query processing phase, servers execute: (1) Star Deposition splits the query Q into balanced star subqueries while minimizing $|q_{\text{max}}|$; (2) Star Matching processes $|q_{\text{max}}|$ using BucketJoin (bucket-partitioned parallel joins on $[\![R(e)]\!]$) and reuses results for smaller subqueries; and (3) Star Assembly greedily merges subqueries to produce the final output $[\![R(Q)]\!]$, then performs symmetry verification before sending the results to the data analyst for reconstruction. Note that all join operations involve four steps: secure join, estimating upper bounds, oblivious compaction (OPAC), and injectivity verification.**

efficient query processing, the data owner performs degree counting on the data graph, selects the maximum degree as the maximum frequency of $R(e)$, and releases *differentially private maximum frequency* (DPMF). The data owner then generates differentially private hierarchical histograms based on the degree statistics, and computes the *differentially private index* (DPIdx) through range queries on the hierarchical histograms (detailed in Section 4).

Online Query Processing. As illustrated in the left part of Figure 1, the query processing phase consists of three main steps.

(1) **Star Decomposition.** Upon receiving the query graph Q submitted by the data analyst, the servers decompose Q into a minimal set of stars, followed by balanced processing to minimize the size of the largest star (see Section 5.1).
(2) **Star Matching.** The biggest star subgraph matches, with the star's root serving as the common join key for each round of joins. The continuous update of the join key DPIdx in each round ensures its persistent effectiveness, enabling data to be bucketed and joined in parallel according to DPIdx. Meanwhile, the topology reuse mechanism generates matching results for all smaller star subqueries (see Section 5.2).
(3) **Star Assembly.** The servers join star subquery results in ascending order of estimated join output size from DPMF, assembling them into secret-shared final query results $R(Q)$, and employ the Symmetry Verification protocol to filter out redundant results (see Section 5.3). These results are subsequently transmitted back to the data analyst for local reconstruction into plaintext.

Note. All secure join results in the PRISM are processed by integrating several key components. First, the DPMF is used to estimate the Hybrid Bound of the join, which shrinks the original results by the oblivious compaction protocol (OPAC). Subsequently, the Injectivity Verification protocol filters out matching results that do not satisfy the injective constraint. for more details.

4 OFFLINE DATA PREPARATION

In the offline data preparation phase, the data owner transforms the data graph into secret-shared edge match tables and generates differentially private synopses to support secure query processing.

Constructing & Sharing Edge Matches. Before query processing, the data owner first preprocesses the data graph G to construct the edge matches relation $R(e)$, which serves as the join unit for online query processing. Specifically, $R(e)$ is formed by concatenating the edge list of G with its transpose, thereby capturing all possible matches for edge queries. The data owner then secret-shares $R(e)$ among three servers using a replicated secret sharing scheme, which reveals no structural information about the data graph.

Subsequently, the data owner publishes differentially private statistical synopses for attribute values in $R(e)$. Since the two columns of $R(e)$ are symmetric, the synopses are computed from either column. These synopses comprise a DPIdx $\hat{\mathcal{I}}$ for efficient data access and a DPMF $\hat{\mathcal{MF}}$ for estimating join sizes. Separate privacy budgets, ϵ_{idx} and ϵ_{mf}, are allocated to these synopses to ensure privacy. We detail the computation of both synopses as follows.

Computing DPIdx. Existing approaches [23, 41] use flat cumulative histograms for indexing. However, this method accumulates significant noise, especially in later intervals, adversely affecting query accuracy and efficiency.

To mitigate this, PRISM adopts *hierarchical histograms* for building DPIdx, achieving lower noise complexity of $O((\log n)^3)$ than the $O(n)$ complexity of flat histograms, as analyzed in Appendix B. A hierarchical histogram is structured as a binary tree, where leaf nodes represent individual attribute values (or unit bins), and internal nodes aggregate counts from their descendant leaves. Perturbing this structure corresponds to releasing $h = \lceil \log n \rceil$ simultaneous histograms at varying granularities, where n represents the size of the attribute domain.

Given a range query, it can be efficiently answered by using the minimal set of tree nodes that cover the query interval, summing their noisy counts. Constructing the DPIdx involves performing range queries on differentially private hierarchical histograms. Specifically, the start position of each interval is the cumulative count of preceding bins, while the end position is the cumulative count up to the interval's last bin. These computations incur no additional privacy cost due to the post-processing property of DP.

Adding Laplace noise directly to histogram counts may cause the noisy interval boundaries to exclude actual data points. Concretely, the noisy start position may exceed the true start position, or the noisy end position may fall below the true end position. Therefore, we adopt the one-sided Laplace mechanism from [41], denoted as $\text{Lap}^+(\epsilon, \delta)$ and $\text{Lap}^-(\epsilon, \delta)$, constraining noise to strictly non-negative or non-positive values, respectively. This mechanism guarantees (ϵ, δ)-differential privacy and produces a lossless DPIdx, where noisy interval boundaries never exclude true data points, ensuring complete recall during query processing.

Algorithm 1 (Lines 1–10) details the data owner's construction of differentially private hierarchical histograms and DPIdx. For each attribute $A_j \in R(e).A$ with domain $\text{Dom}(A_j)$, the data owner first constructs a true hierarchical histogram $\mathcal{H}$ that counts the frequency of each value in A_j. If $|\text{Dom}(A_j)|$ is not a power of two, dummy bins with zero counts are added to form a complete binary tree structure. Counts in $\mathcal{H}$ are perturbed twice using one-sided Laplace noise scaled by $\frac{\epsilon_{\text{idx}}}{2h}$ to produce lower-bound ($\mathcal{H}^-$) and upper-bound ($\mathcal{H}^+$) histograms. The data owner partitions $\text{Dom}(A_j)$ into p disjoint intervals $\{B_1, \ldots, B_p\}$. For each interval $B_i = [v_i^s, v_i^e]$, the noisy start position lo_i is computed by querying $\mathcal{H}^-$ over $[\min(A_j), v_i^s)$, clipped to at least 0; similarly, the end position hi_i is computed by querying $\mathcal{H}^+$ over $[\min(A_j), v_i^e]$, clipped to at most $|R(e)|$. The data owner releases the DPIdx $\hat{\mathcal{I}}$ comprising intervals $[lo_i, hi_i)$ for all B_i. Therefore, each tuple $t \in R(e)$ with $t.A_j \in B_i$ is indexed into subset $R(e).\hat{\mathcal{I}}_i$. When sorted by A_j, tuples are quickly accessible within these subsets.

Algorithm 1: DP Synopses Generation

Input: Edge matches $R(e) = (A, T)$, any attribute $A_j \in A$, private parameters $\epsilon_{\text{idx}}, \epsilon_{\text{mf}}, \delta$.
Output: DPIdx $\hat{\mathcal{I}}(R(e), A_j)$ and DPMF $\hat{\mathcal{MF}}(R(e), A_j)$.

// Generate hierarchical histograms
1 $\mathcal{H} \leftarrow$ Construct hierarchical histogram for $R(e).A_j$;
2 $\mathcal{H}^- \leftarrow \mathcal{H} + \text{Lap}^-(\frac{\epsilon_{\text{idx}}}{2h}, \delta)$;
3 $\mathcal{H}^+ \leftarrow \mathcal{H} + \text{Lap}^+(\frac{\epsilon_{\text{idx}}}{2h}, \delta)$;
// Build DPIdx
4 $\hat{\mathcal{I}}(R(e), A_j) \leftarrow \varnothing$;
5 $\{B_1, \ldots, B_p\} \leftarrow \text{Partition}(\text{Dom}(A_j))$;
6 **for** $i \leftarrow 1$ *to* p **do**
7 $\quad [v_i^s, v_i^e] \leftarrow B_i$;
8 $\quad lo_i \leftarrow \max(\text{RangeQuery}(\mathcal{H}^-, [\min(A_j), v_i^s)), 0)$;
9 $\quad hi_i \leftarrow \min(\text{RangeQuery}(\mathcal{H}^+, [\min(A_j), v_i^e]), |R(e)|)$;
10 $\quad \hat{\mathcal{I}}(R(e), A_j) \leftarrow \hat{\mathcal{I}}(R(e), A_j) \cup \{(B_i, [lo_i, hi_i))\}$;
// Compute DPMF
11 $MF(R(e), A_j) \leftarrow \max_{v \in \text{Dom}(A_j)} |\{t \in R(e).T : t.A_j = v\}|$;
12 $\hat{\mathcal{MF}}(R(e), A_j) \leftarrow MF(R(e), A_j) + \text{Lap}^+(\epsilon_{\text{mf}})$;
13 **return** $(\hat{\mathcal{I}}(R(e), A_j), \hat{\mathcal{MF}}(R(e), A_j))$;

The right part of Figure 1 provides a concrete example. The data owner computes the frequency of each attribute value in any column of $R(e)$ (*i.e.*, degree statistics) and generates two hierarchical histograms $\mathcal{H}^+$ and $\mathcal{H}^-$ using Lap^+ and Lap^-, where p_i^j (*resp.* m_i^j) denotes the i-th element in the j-th layer from bottom to top in $\mathcal{H}^+$ (*resp.* $\mathcal{H}^-$). The index range of $B_2 = \{u3, u4\}$ is computed as follows: the start position is m_0^1 from range query $\{u1, u2\}$ on $\mathcal{H}^-$, and the end position is p_0^2 from range query $\{u1, u2, u3, u4\}$ on $\mathcal{H}^+$.

Computing DPMF. The DPMF is computed straightforwardly via the Laplace mechanism. The data owner adds noise from $\text{Lap}^+(\epsilon_{\text{mf}})$ directly to the true maximum frequency (Algorithm 1, Lines 11–12).

We choose to separate privacy budgets for indexing (ϵ_{idx}) and maximum frequency (ϵ_{mf}), rather than directly using the maximum leaf count from $\mathcal{H}^+$ as the noisy maximum frequency. This is because using solely the indexing budget ϵ_{idx} results in tiny per-level budgets (since it must be split across $h = \lceil \log n \rceil$ levels), which significantly inflates the noise added to the DPMF and thereby diminishes its utility.

Updating Synopses. Since PRISM involves multi-round joins, timely synopsis updates are critical. For DPIdx, the encrypted table (sorted by the indexed attribute) is partitioned into buckets aligned with index intervals, and joins are performed within buckets (BucketJoin, detailed in Section 5.2). Join results update bucket sizes and index positions. Thus, the DPIdx remains valid only for the join-key attribute utilized by BucketJoin. Indices on non-join-key attributes or those used in basic joins become invalid after joins.

For maximum frequency updates, given a join $R = R_1 \bowtie R_2$, we compute the maximum frequency of the attribute A_j as $\hat{\mathcal{MF}}(R, A_j) = \hat{\mathcal{MF}}(R_1, A_j) \times \hat{\mathcal{MF}}(R_2, A_j)$. This multiplicative update strategy ensures the propagated maximum frequency remains a valid upper bound on the true maximum frequency in the join result, which is essential for maintaining the utility of our DP synopses.

5 ONLINE QUERY PROCESSING

In the online query processing phase, upon receiving a query graph Q, the servers execute a three-step pipeline to compute subgraph matches securely and efficiently.

5.1 Star Decomposition

Since we do not consider the privacy of the query graph Q, the query decomposition is performed by the servers in plaintext. We introduce a decomposition algorithm that breaks the query graph Q into m star subqueries $SQ = \{q_1, \ldots, q_m\}$, where a star subquery contains a *root* and a set of *leaves* as its neighbors in Q. PRISM aims to achieve the following two goals.

- **Goal 1.** The number of decomposed star subqueries should be as small as possible, which reduces the number of joins in the Star Assembly step.
- **Goal 2.** The size of the biggest star should be as small as possible, which reduces the number of joins in the Star Matching step.

To realize **Goal 1**, we address a minimum cover problem: we define SQ as a *star cover* of Q if and only if every edge in Q belongs to only one star $q_i \in SQ$. It can be shown that the minimum star cover problem, within polynomial time, is equivalent to the minimum vertex cover problem, which is known to be NP-hard. As a result, the issue at hand is NP-hard as well. We employ a 2-approximation algorithm [38] to construct a star cover from a vertex cover in polynomial time, as detailed below. In every iteration, the algorithm selects the vertex u with the highest degree as the root of a star, adds that vertex to the answer, and removes all edges incident to u. This process is repeated until no edges remain. We use the same method to create a 2-approximate star cover.

For **Goal 2**, we balance the sizes of the stars on previous results, which is a min-max optimization problem. First, we select the current biggest star $q_{\max}$ from SQ, with its root being u_r. Then, we traverse the leaves of $q_{\max}$ to check if any leaf v serves as the root of another star subgraph q_v. To ensure that moving an edge reduces the overall size of the biggest star, we further verify whether the size difference between $q_{\max}$ and q_v is greater than 1. If so, we move the edge (u_r, v) from $q_{\max}$ to q_v. This process repeats until no further reduction in the size of the biggest star can be achieved.

Algorithm 2 outlines the detailed steps of decomposition, where lines 1-6 describe the greedy decomposition process and lines 7-17 detail the balancing procedure applied to the preliminary decomposition result. By simulating the vertex cover approximation algorithm in greedy decomposition and applying min-max optimization in balance processing, this decomposition algorithm achieves both of the above goals.

Algorithm 2: Star Decomposition

Input: Query graph Q.
Output: Set of decomposed star subqueries SQ.

```
// Greedy decomposition
1  SQ ← ∅, E' ← E(Q);
2  while E' ≠ ∅ do
3      Select u ∈ V(Q) with maximum d'(u) in subgraph Q', where
         V(Q') ← V(Q), E(Q') ← E';
4      Create star q_u ← {(u, v) ∈ E' | v ∈ N(u)};
5      SQ ← SQ ∪ {q_u};
6      E' ← E' − E(q_u);
   // Balance processing
7  while true do
8      reduced ← false;
9      q_max ← Select the biggest star in SQ with root u_r;
10     for each v ∈ V(q_max) and v ≠ u_r do
11         if v is the root of q_v ∈ SQ − {q_max} and |q_max| − |q_v| > 1
             then
12             E(q_max) ← E(q_max) − {(u_r, v)};
13             E(q_v) ← E(q_v) ∪ {(u_r, v)};
14             reduced ← true;
15             break;
16     if !reduced then
17         break;
18 return SQ;
```

5.2 Star Matching

Once the servers have prepared the SQ, they proceed to compute the matches for each star subquery in SQ by performing multi-round joins on the edge matches $R(e)$, which offers two advantages.

- **Consistent indexing.** By using the star root consistently as the join key, the index associated with the root is continuously updated and leveraged, significantly accelerating join operations.
- **Topology reuse.** By reusing matching results for identical topologies, matches for smaller stars are naturally derived during the matching of the largest star, avoiding redundant computations.

BucketJoin. Regarding the first advantage, the servers can utilize the BucketJoin to fully leverage the DPIdx released by the data owner. BucketJoin employs a deterministic partitioning strategy guided by the DPIdx as illustrated in Algorithm 3. Initially, using the fixed join key u_r (root of the biggest star $q_{\max}$), the servers apply the secure sorting function $\mathcal{F}_{\text{Sort}}$ to the join unit $[\![R(e)]\!]$ to obtain $[\![R'(e)]\!]$, and organize tuples into p distinct buckets B_i as specified by the DPIdx $\hat{I}$. For each bucket, the servers invoke EnhancedJoin using u_r as the join key on the corresponding portions of $[\![R'(e)]\!]$ and intermediate result $[\![P]\!]$. EnhancedJoin is a secure protocol that compresses the join result size while ensuring subgraph matching constraints are satisfied, which we detail in Appendix E. Note that we first need to rename the attribute names of $[\![R'(e)]\!]$ based on the currently processed edge. Due to the independence of the bucket operations, these join processes can be executed in parallel, thus improving overall efficiency. Following the bucket-level joins, the results are concatenated sequentially, and the index for join key u_r is updated based on the output size of each bucket.

However, BucketJoin requires strict conditions: valid join-key indices and consistent index intervals across tables. Unlike previous work [41], which has not validated such optimizations in practice, PRISM leverages the centralized structure of the star to maintain a consistent join key and continuously updated indices, thereby satisfying the prerequisites of BucketJoin. Furthermore, since BucketJoin partitions buckets based on the DPIdx, each bucket interval covers all true values, ensuring lossless results. Additionally, due to the post-processing property of DP, using and updating the index does not introduce extra privacy loss.

Algorithm 3: Star Matching

Input: Set of star subqueries SQ, shared relation $[\![R(e)]\!] = ((u, v), [\![T]\!], [\![isNull]\!])$ with DPIdx $\hat{I}$.
Output: A shared star matching vector $[\![RS]\!]$.

1 $[\![R'(e)]\!] \leftarrow \mathcal{F}_{\text{Sort}}([\![R(e)]\!], u)$ where assuming u as the star root;
2 $[\![P]\!] \leftarrow \varnothing$;
3 $q_{\max} \leftarrow$ the biggest star in SQ with size k;
// Initialize storage vector for topology reuse
4 $[\![RS]\!] \leftarrow ([\![R(S_1)]\!], \ldots, [\![R(S_k)]\!])$ where $R(S_i) \leftarrow \varnothing$, and $R(S_i)$ is the matches of the star with size i;
5 **for** *each edge* $(u_r, v) \in E(q_{\max})$ **do**
6 $i \leftarrow$ the partial star size;
7 $[\![R'(e)]\!] \leftarrow \rho_{(u_r,v)}(R'(e))$;
8 $[\![P]\!] \leftarrow \text{BucketJoin}([\![P]\!], [\![R'(e)]\!])$;
// Store intermediate results for reused
9 **if** $\exists q \in SQ$ *such that* $|q| = i$ **then**
10 $[\![R(S_i)]\!] \leftarrow [\![P]\!]$;
11 **return** $[\![RS]\!]$;
12 **Function** BucketJoin($[\![R_1]\!]$, $[\![R_2]\!]$):
13 **for** $i \leftarrow 1$ *to* p **do**
14 $[\![P_i]\!] \leftarrow$ EnhancedJoin($[\![R_1.\hat{I}_i]\!], [\![R_2.\hat{I}_i]\!]$);
15 $[\![P]\!] \leftarrow [\![P_1]\!] \parallel [\![P_2]\!] \parallel \cdots \parallel [\![P_p]\!]$;
16 **return** $[\![P]\!]$;

Algorithm 4: Star Assembly

Input: Set of decomposed star subqueries SQ, a shared star matching vector $[\![RS]\!] = ([\![R(S_1)]\!], \ldots, [\![R(S_k)]\!])$.
Output: The final assembled $[\![R(Q)]\!]$.

1 $[\![R(P)]\!] \leftarrow \varnothing$;
2 **while** $|SQ| > 1$ **do**
3 $b_{\min} \leftarrow \infty$;
4 $(sq_a, sq_b) \leftarrow (\varnothing, \varnothing)$;
5 **for** *each pair* (q_a, q_b) *in* SQ **do**
6 $P \leftarrow$ the subquery merged by q_a and q_b;
7 $b \leftarrow \text{HybridBound}(P)$;
// Find the pair with minimal bound
8 **if** $b < b_{\min}$ **then**
9 $b_{\min} \leftarrow b$;
10 $(sq_a, sq_b) \leftarrow (q_a, q_b)$;
// Topology reuse star matches
11 $[\![R(sq_a)]\!] \leftarrow$ if sq_a is a star then $[\![R(S_{|sq_a|})]\!]$ else $[\![R(P)]\!]$;
12 $[\![R(sq_b)]\!] \leftarrow$ if sq_b is a star then $[\![R(S_{|sq_b|})]\!]$ else $[\![R(P)]\!]$;
13 $[\![R(P)]\!] \leftarrow$ EnhancedJoin($[\![R(sq_a)]\!], [\![R(sq_b)]\!]$);
// Update SQ
14 $SQ \leftarrow SQ - \{sq_a, sq_b\} \cup \{P\}$;
15 $[\![R(Q)]\!] \leftarrow$ SymmetryVerify($[\![R(P)]\!]$);
16 **return** $[\![R(Q)]\!]$;

Topology Reuse Mechanism. Regarding the second advantage, the topology reuse mechanism leverages the fact that identical subgraph topologies yield identical matches, which are reusable. This benefit arises because the considered G is unlabeled, which means that the join queries corresponding to subgraph matches with identical topologies are also similar.

Using this property, we first obtain the biggest star subquery $q_{\max}$ in SQ. Assuming the size of the biggest star $|q_{\max}| = k$, we then adjust the output matching results to store a star matching vector $[\![RS]\!] = ([\![R(S_1)]\!], \ldots, [\![R(S_k)]\!])$ rather than the intuitive matches of each subquery in SQ, where $R(S_i)$ corresponds to the matches of the star with size i, as shown in Algorithm 3. When SQ contains multiple star subqueries of the same size, this design significantly reduces the memory overhead of the output.

Constructing $[\![RS]\!]$ only requires computing the matches of $q_{\max}$, since the partial result $[\![P]\!]$, which represents the matches of a partial star, contains the topology of all smaller stars, and thus simultaneously generates the matches for these smaller star subqueries. As shown in lines 5-10 of Algorithm 3, if there exists a subquery in SQ whose size i is identical to that of the partial star, the algorithm explicitly assigns the partial result $[\![P]\!]$ to the corresponding $[\![R(S_i)]\!]$; otherwise, $[\![R(S_i)]\!]$ remains empty. Consequently, constructing $[\![RS]\!]$ requires only $(k-1)$ rounds of join operations, significantly reducing the overall number of joins.

5.3 Star Assembly

Given a query Q decomposed into star subqueries $SQ = \{q_1, \ldots, q_m\}$, the previous step produces their matching relations, stored securely as $[\![S]\!]$. Here, the servers must join these star matches to assemble the final query results.

Secure join operations are computationally costly, primarily due to the size of intermediate results. Each round's output becomes the input for the subsequent round, significantly impacting efficiency. Thus, we aim to achieve the key objective of identifying a multi-round join order with cost as low as possible.

Optimizing Join Order. To achieve this goal, we adopt a greedy approach that dynamically selects join operations to minimize intermediate result sizes. Algorithm 4 describes our star assembly process, which iteratively merges subqueries from SQ using the previously obtained star matching vector $[\![RS]\!]$.

Starting from an empty intermediate result $[\![R(P)]\!]$, we repeatedly evaluate all possible pairs of subqueries (q_a, q_b), estimating their merged size using the Hybrid Bound, a method that combines DPMF to estimate the upper bound of join results (detailed in Appendix C). The optimal pair with the smallest estimated upper bound is selected for merging. We retrieve their corresponding matching results: if a selected subquery is a star, we directly access its matches from $[\![RS]\!]$; otherwise, we use the current intermediate result. We execute the EnhancedJoin protocol on these chosen matches to update $[\![R(P)]\!]$. Finally, we update the subquery set SQ by removing the merged subqueries and adding the newly created merged subquery P. Once only a single subquery remains in SQ, the intermediate result $[\![R(P)]\!]$ is processed by SymmetryVerify to eliminate redundant symmetric results, producing the final assembled query result $[\![R(Q)]\!]$. SymmetryVerify is a secure protocol for detecting symmetric redundancy in join results, with its detailed design presented in Appendix D. When SQ reduces to a single subquery, the intermediate result $[\![R(P)]\!]$ undergoes SymmetryVerify, producing the final assembled query result $[\![R(Q)]\!]$.

Optimality Discussion. The greedy join order optimization in Algorithm 4 follows a similar principle to optimal greedy algorithms

for hierarchical construction problems. At each iteration, we select the pair of subqueries with the minimum estimated join cost, analogous to selecting minimum-weight elements in classical greedy optimization.

This strategy is theoretically sound because: (1) the join cost exhibits submodular properties where early optimal local decisions contribute to global cost reduction, and (2) our hybrid bound estimation provides accurate cost metrics that preserve the greedy choice property. While global optimality cannot be guaranteed due to the NP-hard nature of general join ordering, the combination of accurate cost estimation and the favorable cost structure induced by star decomposition ensures that our greedy approach achieves near-optimal performance in practice.

5.4 Complexity Analysis

Let N denote the size of the edge matching results $R(e)$. We analyze the time complexity of each phase in the online query processing.

(1) **Star Decomposition.** Star Decomposition scans the query once, identifies the m stars, and records the biggest star $q_{\max}$. Its time complexity is $O(|V(Q)| + |E(Q)|)$. Since $|V(Q)|$ and $|E(Q)|$ are much less than N, this term is treated as $O(1)$ for complexity purposes.

(2) **Star Matching.** This phase contains $(k-1)$ enhanced secure join rounds, where $k = |q_{\max}|$ is the size of the biggest star. According to the analysis in Appendix E, the round i of enhanced secure join has complexity $O(S_{i-1} \log S_{i-1})$, where S_{i-1} represents the input size for round i.

(3) **Star Assembly.** Star Assembly begins with join order optimization, where the greedy algorithm evaluates all possible pairs of subqueries, requiring a polynomial function in m, where $m = |SQ|$. Since m is typically a small query-dependent constant, this contributes $O(1)$ to the overall complexity. Subsequently, the optimized join order executes $(m-1)$ enhanced secure join rounds to merge the star matching results.

Overall Complexity Analysis. Combined with the Star Matching and Star Assembly round, the total number of enhanced secure join rounds is $t = (k-1) + (m-1)$. Denote by $\alpha_i \geq 1$ the per-round expansion factor, that is, the ratio of the Hybrid bound to the input size for round i. To keep the notation compact, we bound all α_i with the maximal expansion factor $\alpha = \max_{1 \leq i \leq t} \alpha_i$ and write the conservative output size of the i-th round as $S_i = N\alpha^i; (0 \leq i < t)$ where the factor α^i should therefore be read as a cumulative expansion bound. Therefore, the total cost of all joins is

$$\sum_{i=1}^{t} O(S_{i-1} \log S_{i-1}) = O\Big(N \log N \sum_{j=0}^{t-1} \alpha^j\Big) = O\big(N\alpha^{t-1} \log N\big). \quad (3)$$

After the last join, SymmetryVerify examines the assembled tuples once and requires $O(|R(Q)|)$ time, where $|R(Q)|$ is the cardinality of the final result. By construction $|R(Q)| \leq S_t \leq N\alpha^t$, it is dominated by the right-hand side of Equation 3.

Consequently, the total running time of the online query processing phase is $O(N\alpha^{t-1} \log N)$. When the Hybrid Bound is tight ($\alpha \approx 1$), the complexity collapses to $O(N \log N)$. In the worst case, without bound estimation ($\alpha \approx N$), it degrades to $O(N^t \log N)$, equivalent to traditional secure join protocols that pad intermediate results to the Cartesian product size to ensure obliviousness.

Table 1: Query workloads

No.	Name	$\|E\|$	Description
Q1	wedge	2	2-edge path with a shared vertex.
Q2	triangle	3	3-vertex cycle.
Q3	square	4	4-vertex cycle.
Q4	K4	6	Complete graph of 4 vertices.
Q5	pentagram	5	5-vertex star-like cycle.
Q6	W5	8	Square with central vertex connected to all.
Q7	K5	10	Complete graph of 5 vertices.

6 EVALUATION

6.1 Experimental Setup

Datasets and Workloads. We employ three real-world unlabeled, undirected graphs: Dolphins [26] (D1, 0.3k edges), Euroroads [34] (D2, 2.8k edges), and Powergrid [47] (D3, 13k edges), which are stored as edge tables. The graph of Dolphins has a significantly higher density than the others. To evaluate scalability, we design seven query graphs, ranging from a simple wedge (2 edges) to a complex 5-clique (10 edges), as listed in Table 1. The dataset scale reflects the overhead imposed by our stronger security model and unrestricted query patterns, which incur rapidly increasing communication rounds as query complexity grows. Nonetheless, our experimental results demonstrate substantial efficiency improvements over prior work with comparable technical approaches.

Metrics. We primarily evaluate performance based on execution time and memory cost. Since some tests fail on complex query graphs due to hardware bottleneck limitations, we introduce the theoretical size of join results as an additional performance metric, which is negatively correlated with performance.

Competitors. Given fundamental differences in security models and result accuracy, traditional privacy-preserving subgraph matching methods cannot serve as direct baselines. Methods like k-automorphism [12, 18, 21] and ball-structure approaches [48] require analyst-side plaintext post-filtering of noisy candidates, fundamentally differing from our setting where exact results are computed entirely within the secure protocol. We construct two baselines under the same 3PC setting as PRISM, using identical secure join protocols to ensure obliviousness but differing in join output size upper-bound estimation. The MPC-only adopts the estimation strategy from SMCQL [7], relying solely on MPC, and the SPECIAL incorporates DP synopses following [41] for efficiency. This design isolates the impact of our DP-accelerated optimizations while maintaining consistent security guarantees.

Implementation. Experiments are performed on an Ubuntu 22.04.5 LTS desktop with an Intel(R) Core(TM) i5-10400F CPU @ 2.90 GHz and 64 GB RAM. Participating nodes are simulated as threads that communicate locally over TCP, resulting in sub-millisecond latency. The codebase is implemented entirely in C++, built with CMake 3.18, and compiled using g++-11. It relies on OpenMP and GLPK libraries. Privacy parameters are set as security parameter $\kappa = 128$, probability $\delta = 0.0001$, total budget $\epsilon = 2.0$, and a default ϵ_{idx} to ϵ_{mf} ratio of 2:3.

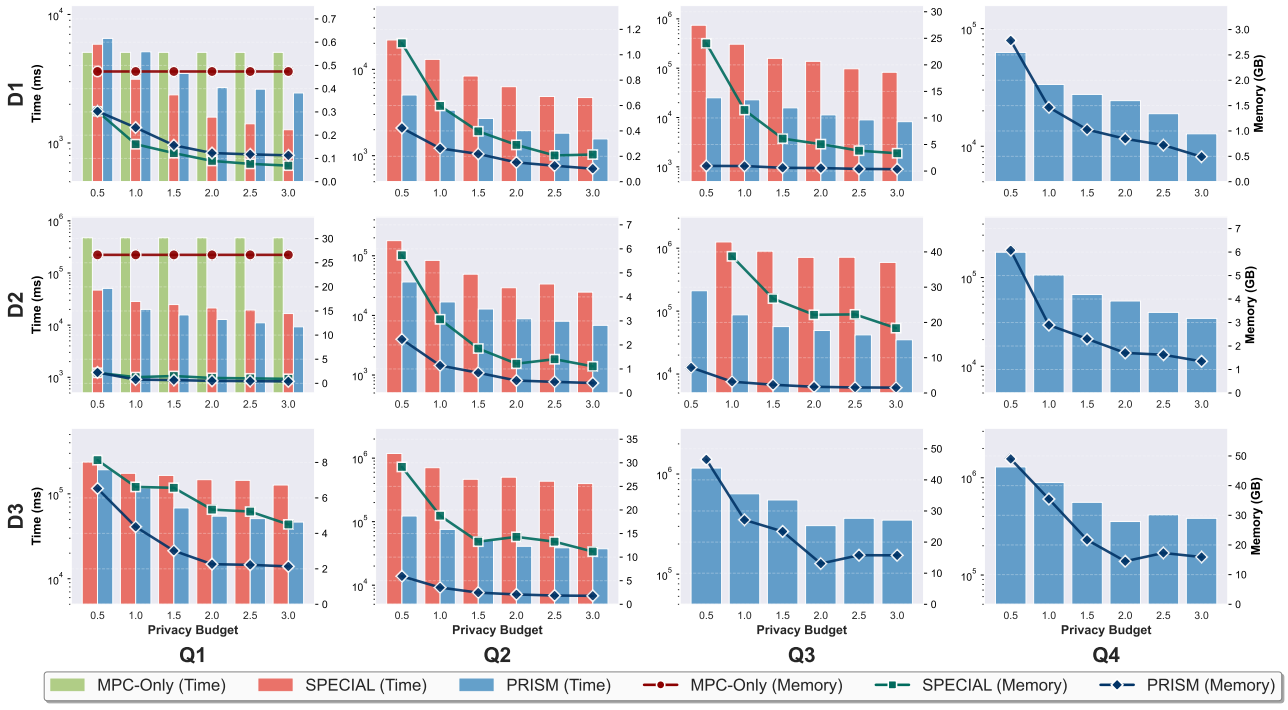

Figure 2: End-to-end evaluation. The columns denote different queries and the rows denote different datasets. Each plot shows the performance of three methods under varying privacy budgets (x-axis), with bars showing time (left y-axis) and lines showing memory (right y-axis). Absent results indicate hardware-induced failures.

6.2 End-to-End Evaluation

As shown in Figure 2, we evaluate end-to-end performance using queries Q1 to Q4 on graphs D1 to D3 with different privacy budgets, measuring execution time and memory consumption. The total privacy budget ϵ ranges from 0.5 to 3.0 in six increments to illustrate the trade-off between privacy and efficiency.

Due to the rapid increase in memory cost with the number of joins, which directly correlates with query graph complexity, we were unable to complete the full testing of baseline methods. For example, MPC-only could only complete the tests of Q1 on D1 and D2; the current state-of-the-art method SPECIAL could not complete the tests beyond Q3. We found that in tests where baseline methods could be completed, even the time and memory overhead of the SOTA method reached 29.6× and 24.9× that of PRISM, respectively. Moreover, as query graph complexity grows and data graph scale increases, the performance gap ratio between the baseline methods and PRISM continues to increase significantly, causing them to quickly reach memory bottlenecks and thus fail to complete testing. For cases where the query graph is very simple and the data graph scale is very small, such as querying Q1 on D1, PRISM's performance may be slightly inferior to the SOTA method due to the non-negligible overhead of PRISM's optimization components. This marginal disadvantage quickly diminishes and reverses as the complexity and scale of the test cases increase.

For different allocations of privacy budget, the performance of both PRISM and SPECIAL generally improves with increasing budget, while MPC-Only remains unaffected as it does not involve DP. Compared to SPECIAL where testing is feasible, PRISM performs with smaller impact from budget variations, allowing for the selection of stricter budget allocations in practical deployment. When the total budget exceeds $\epsilon = 2$, the performance gains begin to plateau. Therefore, we choose $\epsilon = 2$ as the default value for the following evaluations.

6.3 Ablation Studies

DPIdx. To evaluate the hierarchical histogram-based DPIdx, we compare it against a flat histogram cumulative aggregation index on D1 using seven query graphs. Other parameters remain constant.

Figure 3(a) demonstrates consistent performance improvements from the hierarchical histogram-based DPIdx. However, its advantage cannot scale linearly with query complexity, as the index primarily accelerates BucketJoin in star matching. The hybrid upper bound estimation in the Star Assembly step limits its impact.

Join Strategy. We evaluate our star-based join strategy against the baseline left-deep strategy using seven query graphs on D2. Other parameters remain fixed. We measure overall performance and the total number of joins performed.

As shown in Figure 3(b), our star-based join strategy significantly reduces join counts and improves efficiency, while left-deep joins

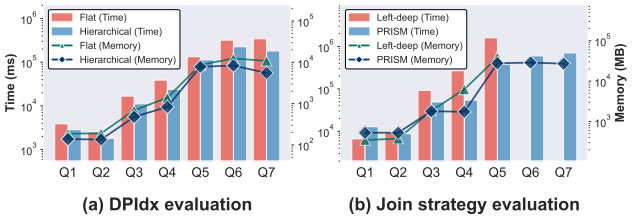

Figure 3: DPIdx and join strategy evaluation.

Table 2: The number of join in different join strategies

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Left-deep	1	2	3	5	4	7	9
PRISM	1	2	2	3	3	4	5

fail beyond Q5. Table 2 further illustrates the growing advantage of our strategy in terms of query complexity, reducing join operations by nearly half.

Upper Bound Estimation. We assess the effectiveness of our Hybrid bound in limiting intermediate result growth by comparing it against the Cartesian product, MF-only and AGM-only on D1 with Q1 to Q7. Parameters aside from upper bound estimation remain unchanged. We record overall execution performance, maximum intermediate, and final output sizes. Due to the exponential growth of intermediate results, the baseline method failed to execute fully; therefore, we estimated its theoretical bounds instead.

As illustrated in Figure 4, the Hybrid bound significantly outperforms existing worst-case estimation methods, successfully executing all tests. The Cartesian product and MF bounds rapidly expand, halting beyond Q1 and Q3, respectively, whereas the AGM bound more accurately predicts final outputs but is hindered by large intermediate sizes, stopping beyond Q4. Based on the theoretical bound, the unoptimized Cartesian product exhibited the fastest growth, reaching $10^{18}\times$ the hybrid bound at Q7, while the MF bound alone reached $10^{7}\times$ and the AGM bound alone achieved 20.6× maximum intermediate during computation. The Hybrid bound effectively mitigates the weaknesses of both AGM and MF bounds, avoiding the rapid expansion seen in MF bound and the excessive intermediate results encountered in AGM bound, demonstrating clear scalability advantages with increasing query complexity.

6.4 Scalability Evaluation

We evaluate the performance of PRISM on queries Q1 to Q7 over datasets D1, D2, and D3 to examine how the overall overhead changes with varying dataset sizes and query complexities. Other parameters remained constant.

As shown in Figure 5, all tests complete successfully on datasets D1 and D2, while D3 (over 4× larger than D2) can only complete up to Q4. Query efficiency exhibits three distinct tiers based on execution time: the first tier includes Q1 and Q2 with similar performance, the second tier comprises Q3 and Q4 with roughly 8× higher cost than the first tier, and the third tier contains Q5, Q6 and Q7 with approximately 8× higher cost than the second tier. This

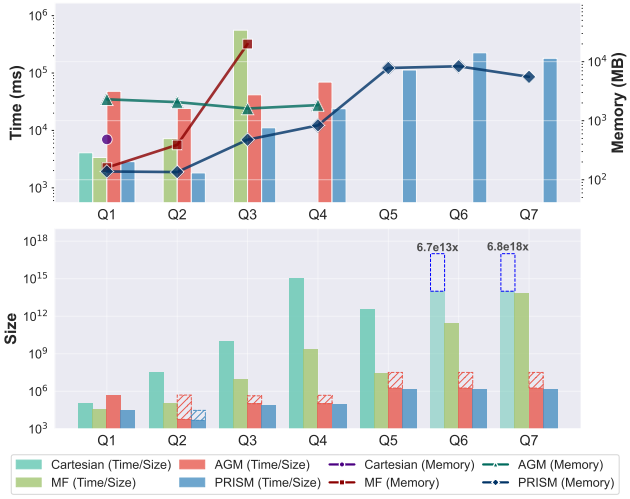

Figure 4: Upper bound evaluation.

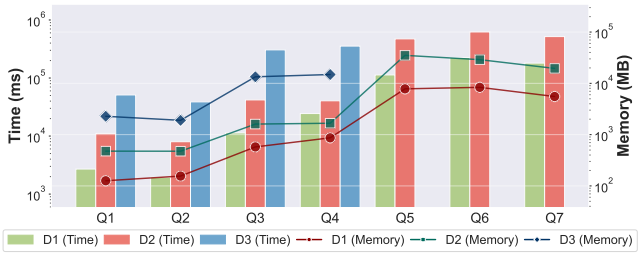

Figure 5: Scalability evaluation.

tiered performance results from the combined effect of upper bound estimation algorithms and join strategies. As query complexity increases, the optimization reduces Q7 and Q6 to costs comparable to Q5 with 5 edges. Therefore, PRISM demonstrates excellent scalability with increasingly pronounced optimization benefits for more complex query graphs.

6.5 Privacy Budget Allocation

We examine the impact of the allocation of ϵ_{idx} to ϵ_{mf} under a fixed total privacy budget ($\epsilon = 2$). This answers why we choose 2:3 as the default privacy budget allocation ratio. To make the effect of the ratio on the query more pronounced, we choose the 4-clique with high utilization across optimization algorithms as the query graph. We increment the ratio by 0.2 steps and evaluate query efficiency and the final output size across all three datasets.

As shown in Table 3, overall efficiency is influenced by both ϵ_{idx} and ϵ_{mf}. Specifically, ϵ_{idx} affects the performance of DPIdx by influencing the size of BucketJoin outputs, while ϵ_{mf} directly impacts the upper bound estimation of the complete query. Experimentally, we find an inflection point of performance around the range (0.8, 1.2), thus we set the default ratio between ϵ_{idx} and ϵ_{mf} to 2:3.

Table 3: Performance comparison on 4-clique queries across different datasets and privacy budget allocation

Dataset	Metric	(0.2, 1.8)	(0.4, 1.6)	(0.6, 1.4)	(0.8, 1.2)	(1.0, 1.0)	(1.2, 0.8)	(1.4, 0.6)	(1.6, 0.4)	(1.8, 0.2)
D1	Time (ms)	59220	49381	31306	22058	17231	14085	12589	**10278**	12547
	Memory (MB)	2324.25	1925.07	1242.78	785.586	727.535	545.906	436.332	**372.742**	410.871
	Final Size	71550	81408	81408	91902	101124	101124	101124	101124	101124
D2	Time (ms)	62665	55041	56845	**54065**	58743	70816	85443	178922	295340
	Memory (MB)	2237.68	1842.16	2050	**1727.7**	2102.24	2584	3047.39	5343.38	10571.1
	Final Size	555464	555464	637650	637650	725504	918217	1133600	2550600	4310514
D3	Time (ms)	512041	546267	488735	**354907**	595325	537002	722050	860757	–
	Memory (MB)	18662.5	21171.3	19343.2	**14859.5**	23763	22017.8	29456.7	35589.7	–
	Final Size	6976452	8242506	7596288	7596288	9614052	8915088	11869200	14361732	–

7 RELATED WORK

7.1 Interaction between DP and MPC

The DP-MPC intersection has emerged as a promising paradigm for privacy-preserving analytics, addressing the trade-off between efficiency and privacy guarantees.

DP for MPC. A primary motivation for integrating DP into MPC systems is to reduce computational overhead. Wang *et al.* [46] propose adding noise to query trajectories to enable efficient filtering before expensive MPC operations. In contrast, some studies [8, 40] leverage DP to minimize padding requirements for intermediate results, thereby reducing subsequent MPC computational burden. Advanced techniques generate privacy-preserving indexes to facilitate more targeted MPC processing, as demonstrated by SPECIAL [41] and Doquet [33]. Longshot [52] generates differentially private flat histograms to replace expensive MPC aggregation queries, achieving performance improvements. Beyond efficiency, DP enhances MPC privacy protection across multiple dimensions. In federated query scenarios, the systems like SAQE [9] apply output perturbation to prevent individual information leakage from query results. For machine learning applications, Truex *et al.* [37] demonstrate how input perturbation can eliminate individual characteristics. Furthermore, DP strengthens access pattern protection [27, 33] and safeguards privacy in dynamic data scenarios [39].

MPC for DP. Conversely, MPC serves as a crucial building block for bridging the utility gap between Local DP (LDP) and Central DP (CDP) models. Systems like Crypt-ϵ [14], the histogram computation protocol [10], and PEA [35] eliminate trusted curator requirements while achieving CDP-level accuracy, effectively providing the "best of both worlds" through secure multi-server architectures.

7.2 Secure Graph Analytics

Secure graph analytics is a critical research area with three main approaches as follows.

Cryptographic Computation. Homomorphic encryption (HE)-based approaches include two-party subgraph matching protocols that allow parties to jointly determine subgraph relationships while protecting input graphs under semi-honest assumptions [50], and frameworks for privacy-preserving localized graph pattern queries in cloud environments using trusted execution environments [48]. MPC techniques have also been explored, including frameworks that employ garbled circuits to enable parallel secure execution of graph algorithms [30] and secret sharing approaches for the secure computation of breadth-first search and maximal independent set algorithms [3]. Additionally, k-nearest neighbor (kNN) based methods using secure inner product computation for encrypted graph query processing in cloud settings have been proposed [11].

Graph Structure Obfuscation. The k-automorphism model represents a dominant approach to protecting structural privacy. Several studies have leveraged k-automorphic transformations, combined with label generalization, to enable efficient subgraph matching while preserving privacy [12, 18]. This approach was enhanced through the proposal of the (k, t)-privacy model, which combines k-automorphism with t-closeness for stronger privacy guarantees [21]. Building upon the limitations of k-automorphism, researchers introduced k-decomposition algorithms designed for massive graph datasets with improved scalability [16]. Additionally, an ϵ-edge DP approach, PrivGraph [51], leverages community structure by privately aggregating and perturbing intra-community and inter-community edge counts to generate graphs that preserve structural properties.

Hybrid Approaches. Recent works on attributed graphs employ k-automorphism to protect the graph structure while using MPC to secure vertex and edge information [43, 44]. PrivAGM [42] extends this line of work by constructing differentially private directed attributed graph models on decentralized social graphs through selective MPC and hierarchical DP mechanisms.

8 CONCLUSION

We present PRISM, a new MPC-based approach accelerated by DP for secure subgraph matching on outsourced sensitive data. PRISM leverages differentially private synopses to accelerate joins through hierarchical histogram-based indexing and hybrid upper-bound estimation. Furthermore, we design a star-based join strategy that reduces join operations for complex queries. We implement two secure constraint verification protocols, enabling stronger privacy protection than existing approaches while producing more accurate results. Our experiments demonstrate substantial performance improvements across query complexities and graph scales. While SOTA methods only handle simple query tests with 29.6× time and 24.9× memory overhead compared to our approach, PRISM efficiently processes complex queries. Future work can be extended to federated graph settings for secure distributed subgraph matching.

REFERENCES

[1] Airbnb on AWS. 2018. Aws Case Study: Airbnb. https://aws.amazon.com/solutions/case-studies/airbnb/?nc1=h_ls. Accessed: Nov. 15, 2022.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*. 805–817.

[3] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *ACM CCS*. 610–629.

[4] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *ACM CCS*. 125–138.

[5] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *IEEE FOCS*. 739–748.

[6] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *ACM CCS*. 209–222.

[7] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *PVLDB* 10, 6 (2017), 673–684.

[8] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. ShrinkWrap: Efficient SQL Query Processing in Differentially Private Data Federations. *PVLDB* 12, 3 (2018), 307–320.

[9] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-Preserving Approximate Query Processing for Data Federations. *PVLDB* 13, 11 (2020), 2691–2705.

[10] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *ACM CCS*. 307–321.

[11] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. 2011. Privacy-Preserving Query over Encrypted Graph-Structured Data in Cloud Computing. In *IEEE ICDCS*. 393–402.

[12] Zhao Chang, Lei Zou, and Feifei Li. 2016. Privacy Preserving Subgraph Matching on Large Graphs in Cloud. In *ACM SIGMOD*. 199–213.

[13] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, Vol. 6477. 577–594.

[14] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Crypt?: Crypto-Assisted Differential Privacy on Untrusted Servers. In *ACM SIGMOD*. 603–619.

[15] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE TPAMI* 26, 10 (2004), 1367–1372.

[16] Xiaofeng Ding, Cui Wang, Kim-Kwang Raymond Choo, and Hai Jin. 2021. A Novel Privacy Preserving Framework for Large Scale Graph Data Publishing. *IEEE TKDE* 33, 2 (2021), 331–343.

[17] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *PVLDB* 3, 1 (2010), 264–275.

[18] Jiuru Gao, Jiajie Xu, Guanfeng Liu, Wei Chen, Hongzhi Yin, and Lei Zhao. 2018. A Privacy-Preserving Framework for Subgraph Pattern Matching in Cloud. In *DASFAA*, Vol. 10827. 307–322.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM STOC*. 218–229.

[20] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *CIDR*.

[21] Kai Huang, Haibo Hu, Shuigeng Zhou, Jihong Guan, Qingqing Ye, and Xiaofang Zhou. 2022. Privacy and Efficiency Guaranteed Social Subgraph Matching. *VLDBJ* 31, 3 (2022), 581–602.

[22] Yannis E. Ioannidis and Younkyung Cha Kang. 1991. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *ACM SIGMOD*. 168–177.

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *ACM SIGMOD*. 489–504.

[24] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure Collaborative Analytics in Untrusted Clouds. In *NSDI*. 1031–1056.

[25] Yuhan Liu, Hong Chen, Yixuan Liu, Dan Zhao, and Cuiping Li. 2022. State-of-the-art Privacy Attacks and Defenses on Graphs. *Chin J Comput* 4 (2022), 702–734.

[26] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. 2003. The Bottlenose Dolphin Community of Doubtful Sound Features a Large Proportion of Long-lasting Associations: Can Geographic Isolation Explain This Unique Trait? *Behavioral Ecology and Sociobiology* 54, 4 (2003), 396–405.

[27] Sahar Mazloom and S. Dov Gordon. 2018. Secure Computation with Differentially Private Access Patterns. In *ACM CCS*. 490–507.

[28] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*. 35–52.

[29] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *ACM CCS*. 1271–1287.

[30] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S&P*. 377–394.

[31] Xiaofeng Ouyang, Liang Hong, and Lujia Zhang. 2018. Query Associations Over Big Financial Knowledge Graph. In *BigSDM*, Vol. 11473. 199–211.

[32] Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. 2011. Using Graph Theory to Analyze Biological Networks. *BioData Min.* 4 (2011), 10.

[33] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. 2023. Doquet: Differentially Oblivious Range and Join Queries with Private Data Structures. *PVLDB* 16, 13 (2023), 4160–4173.

[34] Ryan Rossi and Nesreen Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[35] Wenqiang Ruan, Mingxin Xu, Wenjing Fang, Li Wang, Lei Wang, and Weili Han. 2023. Private, Efficient, and Accurate: Protecting Models Trained by Multi-party Learning with Differential Privacy. In *IEEE S&P*. 1926–1943.

[36] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB* 5, 9 (2012), 788–799.

[37] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A Hybrid Approach to Privacy-Preserving Federated Learning. In *ACM AISec*. 1–11.

[38] Vijay V. Vazirani. 2001. *Approximation Algorithms*. Springer.

[39] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. DP-Sync: Hiding Update Patterns in Secure Outsourced Databases with Differential Privacy. In *ACM SIGMOD*. 1892–1905.

[40] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy. In *ACM SIGMOD*. 818–832.

[41] Chenghong Wang, Lina Qiu, Johes Bater, and Yukui Luo. 2024. SPECIAL: Synopsis Assisted Secure Collaborative Analytics. *PVLDB* 18, 4 (2024), 1035–1048.

[42] Songlei Wang, Yifeng Zheng, Xiaohua Jia, and Haibo Hu. 2025. PrivAGM: Secure Construction of Differentially Private Directed Attributed Graph Models on Decentralized Social Graphs. *PVLDB* 18, 11 (2025), 4682–4694.

[43] Songlei Wang, Yifeng Zheng, Xiaohua Jia, Hejiao Huang, and Cong Wang. 2022. OblivGM: Oblivious Attributed Subgraph Matching as a Cloud Service. *IEEE TIFS* 17 (2022), 3582–3596.

[44] Songlei Wang, Yifeng Zheng, Xiaohua Jia, and Cong Wang. 2024. eGrass: An Encrypted Attributed Subgraph Matching System with Malicious Security. *IEEE TIFS* 19 (2024), 5999–6014.

[45] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *ACM SIGMOD*. 1969–1981.

[46] Yuxiang Wang, Yuxiang Zeng, Shuyuan Li, Yuanyuan Zhang, Zimu Zhou, and Yongxin Tong. 2024. Efficient and Private Federated Trajectory Matching. *IEEE TKDE* 36, 12 (2024), 8079–8092.

[47] Duncan J Watts and Steven H Strogatz. 1998. Collective Dynamics of 'Small-world' Networks. *Nature* 393, 6684 (1998), 440–442.

[48] Lyu Xu, Byron Choi, Yun Peng, Jianliang Xu, and Sourav S. Bhowmick. 2023. A Framework for Privacy Preserving Localized Graph Pattern Query Processing. *Proc. ACM Manag. Data* 1, 2 (2023), 129:1–129:27.

[49] Lyu Xu, Jiaxin Jiang, Byron Choi, Jianliang Xu, and Sourav S. Bhowmick. 2021. Privacy Preserving Strong Simulation Queries on Large Graphs. In *IEEE ICDE*. 1500–1511.

[50] Zifeng Xu, Fucai Zhou, Yuxi Li, Jian Xu, and Qiang Wang. 2019. Privacy-Preserving Subgraph Matching Protocol for Two Parties. *IJFCS* 30, 4 (2019), 571–588.

[51] Quan Yuan, Zhikun Zhang, Linkang Du, Min Chen, Peng Cheng, and Mingyang Sun. 2023. PrivGraph: Differentially Private Graph Data Publication by Exploiting Community Information. In *USENIX Security*. 3241–3258.

[52] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2023. Longshot: Indexing Growing Databases using MPC and Differential Privacy. *PVLDB* 16, 8 (2023), 2005–2018.

[53] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. 2011. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB* 4, 8 (2011), 482–493.

A (2,3)-REPLICATED SECRET SHARING SCHEME

Assume a secret x is shared as three random elements x_1, x_2, and x_3 that satisfy $x_1 \oplus x_2 \oplus x_3 = x$, and the party P_i holds shares (x_i, x_{i+1}). The (2,3)-replicated secret sharing scheme offers the following efficient operations:

(1) Operation of a single shared value.
   - $\neg[\![x]\!]$: P_1 computes and stores $(\neg x_1, x_2)$; P_2 stores (x_2, x_3); P_3 computes and stores $(\neg x_1, x_3)$.
(2) Operations between a shared value and a plaintext α.
   - $\alpha \wedge [\![x]\!]$: Each P_i computes and stores $(\alpha \wedge x_i, \alpha \wedge x_{i+1})$.
   - $\alpha \oplus [\![x]\!]$: P_1 computes and stores $(\alpha \oplus x_1, x_2)$; P_2 stores (x_2, x_3); P_3 computes and stores $(\alpha \oplus x_1, x_3)$.
(3) Operations between shared values.
   - $[\![x]\!] \oplus [\![y]\!]$: Each P_i computes and stores $(x_i \oplus y_i, x_{i+1} \oplus y_{i+1})$.
   - $[\![x]\!] \wedge [\![y]\!]$: Let $z = x \wedge y$, each P_i computes $z_i = (x_i \wedge y_i) \oplus (x_i \wedge y_{i+1}) \oplus (x_{i+1} \wedge y_i)$; P_i sends z_i to P_{i-1}, and then stores (z_i, z_{i+1}).

Other operations could derive from NOT, XOR, and AND, incurring similar communication complexity. For example, the OR can be expressed as $x \vee y = \neg(\neg x \wedge \neg y)$, which incurs the same communication cost as the AND.

B NOISE ANALYSIS OF DPIDX

Without loss of generality, we compare noise accumulation in simple versus hierarchical histograms under the same total privacy budget ϵ, using the Laplace mechanism. In a flat histogram, Laplace noise drawn from $\mathrm{Lap}(1/\epsilon)$ is added to each unit bin, resulting in a noise variance of $V_u = 2/\epsilon^2$. This mechanism satisfies ϵ-DP since adding or removing a tuple affects at most one bin by 1. When computing an interval's starting position by summing the noisy counts of all previous bins, the variance becomes $|r|V_u$, where $|r|$ is the number of preceding bins. On average, $|r| = \frac{n(n-1)}{2}$, resulting in an average variance of $\frac{(n-1)}{2}V_u$. In the worst case (the final interval), the noise variance reaches $n \cdot V_u$.

For a hierarchical histogram, we assume that the budget is equally allocated among these h histograms, the noise variance in each node is $\frac{2h^2}{\epsilon^2} = h^2 V_u$ with $V_u = \frac{2}{\epsilon^2}$. Due to the binary tree structure, any range query can be represented using at most $2h$ nodes (2 per level), so the worst-case noise variance is bounded by

$$2h \cdot h^2 V_u = 2(\log n)^3 V_u. \tag{4}$$

This is a significant asymptotic improvement over the flat histogram's worst-case variance $n \cdot V_u$, as it not only reduces noise variance but also distributes noise more evenly across intervals, mitigating accumulation issues.

C HYBRID UPPER BOUND ESTIMATION

Secure joins differ from plaintext joins by padding outputs with dummy tuples to prevent information leakage, typically up to the size of the Cartesian product. Although it ensures security, this padding introduces substantial overhead. To address this, we propose a hybrid upper bound estimation strategy coupled with oblivious compaction. Specifically, we first execute the secure join and then apply oblivious compaction to reduce the output size without data loss.

We integrate two complementary upper bound estimation methods: the AGM bound [5] and the MF bound [20] with the following definitions.

Definition C.1 (AGM Bound). The AGM (Atserias-Grohe-Marx) bound estimate the worst-case upper bound for a multi-round join query based on its hypergraph representation. Consider relation $R_1, \ldots, R_m$, each associated with an attribute set A_i. The hypergraph $H = (V, E)$ has vertices V as attributes and hyperedges $E = \{A_1, \ldots, A_m\}$. Formally, the AGM bound is defined as:

$$|R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m| \leq \max \prod_{i=1}^{m} |R_i|^{x_i} \tag{5}$$

subject to linear constraints:

$$\forall v \in V, \sum_{i:v \in A_i} x_i \geq 1, x_i \geq 0. \tag{6}$$

Definition C.2 (MF Bound). The MF (Maximum Frequency) bound estimates the upper bound for a join query based on the maximum frequency of the join key. Given two relations R and S joining on attributes x, y, the MF bound is computed as:

$$|R \bowtie S| \leq \min\left(\frac{|R|}{MF(R.x)}, \frac{|S|}{MF(S.y)}\right) \cdot MF(R.x) \cdot MF(S.y), \tag{7}$$

where $MF(R.s)$ is the maximum frequency of attribute x in R.

The AGM bound exploits query structural properties, typically providing tighter bounds for complex (dense) query graphs. However, for simpler structures (*e.g.*, star or chain queries), its bound approaches the size of the Cartesian product. Conversely, the MF bound relies on maximum frequency statistics of join keys, performing well under uniform distributions but poorly under skewed distributions (power-law).

Our hybrid strategy, detailed in Algorithm 5, recursively computes a triple of estimates (b_A, b_M, b_S) for each intermediate join, selecting the smallest as the upper bound. Specifically, b_A is the AGM bound computed directly from join units representing edge matches, mirroring the query structure. b_M is the MF bound, updated dynamically based on join key frequencies and intermediate result sizes. b_S is an AGM bound computed from a hypergraph consisting of current input relations. The calculation of b_S leverages the *shrinkage property* of the AGM bound: when the hypergraph introduces no new nodes compared to the previous step, the bound from the current step directly constrains the last bound. Since b_M monotonically increases, we only compare b_S and b_A in such scenarios.

Different join orders yield varying upper bound estimates for identical queries. Thus, we exhaustively explore all possible join sequences to identify the tightest upper bound estimate.

The time complexity of Hybrid Bound estimation is a polynomial function of the number of relations, which is typically small and is treated as a constant in this work, and the overall complexity is $O(1)$.

Example. Consider two join trees of the example query K_4 in Figure 6 where case (a) is the star assembly process and case (b) is the left-deep join plan. Assume each edge matching is considered as a join unit with size N and maximum frequency mf. The computation proceeds as follows. In case (a):

- **step 1:** $R(q_{v_1}) = R(e(v_1, v_2)) \bowtie R(e(v_1, v_4))$:

Algorithm 5: Hybrid Bound

Input: The join query q.
Output: The hybrid bound b for query q.
Assume: The match result size $N = |R(e)|$ and its maximum frequency $mf = \hat{\mathcal{MF}}(R(e), u)$.

$b \leftarrow \infty$;
for *each valid and non-redundant join tree* t *of* q **do**
  $(b_H, mf') \leftarrow$ computeHybridBound(t, N, mf);
  $b \leftarrow \min(b, b_H)$;
return b;

Function computeHybridBound(t, N, mf):
  if t *is leaf* **then**
    return (N, mf);
  $(b_l, mf_l) \leftarrow$ computeHybridBound(t.left, N, mf);
  $(b_r, mf_r) \leftarrow$ computeHybridBound(t.right, N, mf);
  if *the left and right children of* t *satisfies shrinkage property* **then**
    $b_M \leftarrow \infty$;
  else
    $b_M \leftarrow$ MFBound(b_l, b_r, mf_l, mf_r);
  $b_A \leftarrow$ GlobalAGMBound(t, N);
  $b_S \leftarrow$ StepwiseAGMBound(t, b_l, b_r);
  return $(\min(b_A, b_M, b_S), mf_l \cdot mf_r)$;

Function GlobalAGMBound(t, N):
  $HG \leftarrow$ the hypergraph of the query represented by the root of t joined by edges;
  return $AGMBound(HG, N)$;

Function StepwiseAGMBound(t, b_l, b_r):
  $HG \leftarrow$ the hypergraph of the query joined by the left and right children of t;
  return $AGMBound(HG, (b_l, b_r))$;

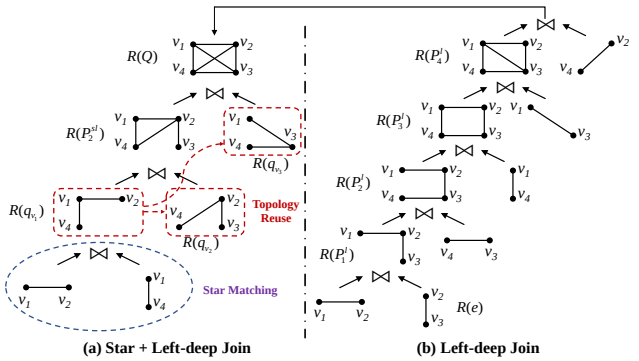

Figure 6: Different Join Trees.

  - $b_A = b_S = N^2$, $w(v_1, v_2) = w(v_1, v_4) = 1$.
  - $b_M = mf \cdot N$.
  - $|R(q_{v_1})| = b_M = mf \cdot N$, due to $mf \leq N$.
- **step 2:** $R(P_2^{sl}) = R(q_{v_1}) \bowtie R(q_{v_2})$:
  - $b_A = N^2$, $w(v_1, v_4) = w(v_2, v_3) = 1$ and others are 0.
  - $b_M = mf^2 \cdot |R(q_{v_1})| = mf^3 \cdot N$.
  - $b_S = |R(q_{v_1})| \cdot |R(q_{v_2})| = mf^2 \cdot N^2 \geq b_A$, $w(v_1, v_2, v_4) = w(v_2, v_3, v_4) = 1$.
  - $|R(P_2^{sl})| = \min(N^2, mf^3 \cdot N)$.
- **step 3:** $R(Q) = R(P_2^{sl}) \bowtie R(q_{v_3})$:
  - $b_A = N^2$, $w(v_1, v_4) = w(v_2, v_3) = 1$, others are 0.
  - b_M is not necessary because of the shrinkage property.
  - $b_S = |R(P_2^{sl})| = \min(N^2, mf^3 \cdot N)$, $w(v_1, v_2, v_3, v_4) = 1$ and others are 0.

In case (b):

- **step 1:** $R(P_1^l) = R(e(v_1, v_2)) \bowtie R(e(v_2, v_3))$:
  - $b_A = b_S = N^2$, $w(v_1, v_2) = w(v_2, v_3) = 1$.
  - $b_M = mf \cdot N$.
  - $|R(P_1^l)| = b_M = mf \cdot N$, due to $mf \leq N$.
- **step 2:** $R(P_2^l) = R(P_1^l) \bowtie R(e(v_3, v_4))$:
  - $b_A = N^2$, $w(v_1, v_2) = w(v_3, v_4) = 1$ and others are 0.
  - $b_M = mf \cdot |R(P_1^l)| = mf^2 \cdot N$.
  - $b_S = |R(P_1^l)| \cdot |R(e(v_3, v_4))| = mf \cdot N^2 \geq b_A$, $w(v_1, v_2, v_4) = w(v_2, v_3, v_4) = 1$.
  - $|R(P_2^{sl})| = \min(N^2, mf^2 \cdot N)$.
- **step 3:** $R(Q) = R(P_4^l) \bowtie R(e(v_2, v_4))$:
  - $b_A = N^2$, $w(v_1, v_4) = w(v_2, v_3) = 1$, others are 0.
  - b_M is not necessary because of the shrinkage property.
  - $b_S = |R(P_2^l)| = \min(N^2, mf^2 \cdot N)$ according to the iterative shrinkage property.

In summary, the hybrid bounds of $R(Q)$ derived from two join trees are $\min(N^2, mf^3 \cdot N)$ and $\min(N^2, mf^2 \cdot N)$. For $mf < N^{0.5}$, the left-deep join tree produces the tighter bound $mf^2 \cdot N$. Otherwise, the bounds of both are the same N^2.

D SECURE CONSTRAINT VERIFICATIONS

Recall our earlier definition of subgraph matching: Given $R(q)$ as the matching result table of graph q, attributes $V(q)$ represent the mappings from $V(q)$ to $V(G)$ in each tuple. However, simply using join operations does not yield accurate results. The fundamental reason for this inaccuracy is that the join cannot constrain values beyond the join key, a fact often overlooked in previous studies. In this section, we analyze these limitations and propose an MPC protocol to implement the ideal constraint verification functionality $\mathcal{F}_{\text{CV}}$. There are two fundamental causes as follows:

(1) While join equality conditions ensure consistency in edge relationships, joins cannot guarantee injectivity since they cannot check for duplicates beyond join keys, resulting in incorrect matches in $R(q)$.
(2) Joins do not detect permutations among tuples. When the query graph q is automorphic, redundant matches in a factorial relationship with symmetric node counts significantly increase the subsequent computational overhead for $R(q)$.

To address these issues, we propose integrating two secure constraint verification protocols, InjectivityVerify and SymmetryVerify, into the secure join process. Specifically, InjectivityVerify is executed after each original secure join protocol to enforce injectivity constraints, while SymmetryVerify is applied only once at the end, directly on the final computed result $[\![R(Q)]\!]$, to ensure no matching results are lost. Both verification protocols operate on intermediate shared matching results $[\![R(q)]\!]$, generated

Protocol 2: Secure Symmetry Verification

Notation: $[\![R_q]\!] \leftarrow \textsc{SymmetryVerify}([\![R_q]\!])$.
Input: Shared matches $[\![R_q]\!]$ where q is a subquery of Q and $|V(q)| = m$.
Output: $[\![R_q]\!]$ with an updated $isNull$.
Assume: $C_{\text{ord}} : \mathbb{K}^m \to 0, 1$ outputs 0 iff the m elements from $\mathbb{K}$ are in ascending order, set of symmetric vertex groups $\text{Svg}(Q) = \{C_1, \ldots, C_k\}$.

for *each match* $[\![t]\!] = ([\![a_1]\!], \ldots, [\![a_m]\!]) \in R_q$ **do**
  $[\![C']\!] \leftarrow \{[\![C'_1]\!], \ldots, [\![C'_k]\!]\}$ where $[\![C'_j]\!] = ([\![a_i]\!] \mid v_i \in C_j \cap V(q))$;
  $[\![e]\!] \leftarrow 0$;
  for *each* $[\![C'_i]\!] \in [\![C']\!]$ **do**
    The servers send $(C_{\text{ord}}, [\![C'_i]\!])$ to $\mathcal{F}_{\text{MPC}}$, and receive $[\![e_i]\!]$;
    $[\![e]\!] \leftarrow [\![e]\!] \vee [\![e_i]\!]$;
  $[\![isNull]\!] \leftarrow [\![isNull]\!] \vee [\![e]\!]$ for tuple t;
return $[\![R_q]\!]$;

Protocol 3: Enhanced Secure Join

Notation: $[\![R]\!] \leftarrow \textsc{EnhancedJoin}([\![R_1]\!], [\![R_2]\!])$.
Input: Shared relation tables $[\![R_1]\!], [\![R_2]\!]$.
Output: Shared join result $[\![R]\!]$.

The servers send $([\![R_1]\!], [\![R_2]\!])$ to $\mathcal{F}_{\text{Join}}$, and receive $[\![R_r]\!]$;
$[\![R_c]\!] \leftarrow \textsc{OblivCompaction}([\![R_r]\!], \text{HybridBound}([\![R_1]\!], [\![R_2]\!]))$;
$[\![R]\!] \leftarrow \textsc{InjectivityVerify}([\![R_c]\!])$;
return $[\![R]\!]$;

Protocol 1: Secure Injectivity Verification

Notation: $[\![R_q]\!] \leftarrow \textsc{InjectivityVerify}([\![R_q]\!])$.
Input: Shared matches $[\![R_q]\!]$ where q is a subquery of Q and $|V(q)| = m$.
Output: $[\![R_q]\!]$ with an updated $isNull$.
Assume: $C_{\text{eq}} : \mathbb{K}^m \to \{0, 1\}^m$ maps vector $(a_1, \ldots, a_m)$ to $(1, b_2, \ldots, b_m)$, where for every $i \geq 2$, $b_i = 1$ iff $a_{i-1} = a_i$.

for *each match* $[\![t]\!] = ([\![a_1]\!], \ldots, [\![a_m]\!]) \in R_q$ **do**
  The servers send $[\![t]\!]$ to $\mathcal{F}_{\text{Sort}}$, and receive $[\![t']\!]$;
  The servers send $(C_{\text{eq}}, [\![t']\!])$ to $\mathcal{F}_{\text{MPC}}$, and receive $[\![D]\!]$;
  $[\![d]\!] \leftarrow \bigvee_{i=1}^{m} [\![D_i]\!]$;
  $[\![isNull]\!] \leftarrow [\![isNull]\!] \vee [\![d]\!]$ for tuple t;
return $[\![R_q]\!]$;

during the secure subgraph matching procedure, where q is a subquery of Q. Additionally, SymmetryVerify leverages precomputed symmetric vertex groups associated with the query graph Q.

In injectivity verification, shown in Protocol 1, for each tuple $[\![t]\!]$ in the matches, the servers first invoke $\mathcal{F}_{\text{Sort}}$ to obtain a sorted tuple $[\![t']\!]$. Next, the servers invoke $\mathcal{F}_{\text{MPC}}$ with circuit C_{eq} to compare adjacent elements in $[\![t']\!]$. For every position $i \geq 2$, if the $(i-1)$-th equals the i-th element, the corresponding output bit is set to 1. A shared flag $[\![d]\!]$ is computed by performing a logical OR over all these shared bits. If $d = 1$, it indicates that t contains duplicate elements and therefore does not form an injective mapping from $V(q)$ to $V(G)$.

The time complexity of InjectivityVerify is $O(|R_q| \cdot |V(Q)| \cdot \log |V(Q)|)$, where $|R_q|$ is the number of intermediate tuples and $|V(Q)|$ is the number of vertices in the query graph. This complexity stems from sorting each tuple of size $|V(Q)|$ and performing adjacent element comparisons. Since $|V(Q)|$ is typically small and is treated as a constant in practice, the overall complexity simplifies to $O(|R_q|)$.

In symmetry verification, shown in Protocol 2, the protocol utilizes the precomputed set of symmetric vertex groups $\text{Svg}(Q) = \{C_1, \ldots, C_k\}$ to divide the attribute values in tuple $[\![t]\!]$ corresponding to q into a set of vectors $[\![C']\!]$ according to each symmetric group in $\text{Svg}(Q)$. Since $\text{Svg}(Q)$ remains invariant for a given query, we treat it as a global variable in our protocol design without explicitly passing it as an input. For each vector, the servers invoke $\mathcal{F}_{\text{MPC}}$ with circuit C_{ord} to verify whether these values adhere to a fixed partial order, for example, by arranging them in ascending order to break symmetry. The circuit C_{ord} detects any incorrect ordering of the vector and sets the shared flag $[\![e]\!]$ to 1.

The time complexity of SymmetryVerify is $O(|R(Q)| \cdot k \cdot m \log m)$, where $|R(Q)|$ is the number of tuples in the final result, k is the number of symmetric groups, and m is the maximum size of symmetric groups. This complexity arises from processing each tuple through k symmetric groups, with each group requiring $O(m \log m)$ operations for the ordering verification circuit. In this work, we treat k and m as constants, resulting in an overall complexity of $O(|R(Q)|)$.

Finally, the servers update the corresponding field $[\![isNull]\!]$ in the $[\![R(q)]\!]$ with a logical OR on the shared flags $[\![d]\!]$ and $[\![e]\!]$, respectively. Together, these two verification protocols ensure that the final matches are injective and redundancy-free. This mechanism significantly improves matching accuracy and query efficiency.

E ENHANCED SECURE JOIN

In our query processing phase, all secure join protocols are replaced with EnhancedJoin, as shown in Protocol 3, which provides improvements in both join efficiency and result accuracy through the components introduced earlier.

For efficiency, we leverage Hybrid Bound to compress the size of each join result across multiple rounds, making the input scale for subsequent joins manageable compared to traditional secure join outputs. We augment the secure join with the protocol OblivCompaction to securely compress secret-shared results. Specifically, given a shared relation $[\![R]\!]$ and an upper bound b, it first invokes $\mathcal{F}_{\text{Sort}}$ to sort $[\![R]\!]$ based on $isNull$, moving tuples with $isNull = 0$ to the front. We then truncate the result to the first b tuples to compress the output to a more compact size.

For result accuracy, we employ the protocol InjectivityVerify to securely validate that each tuple in the join results satisfies the injectivity requirement of subgraph matching. Note that we only apply SymmetryVerify after the final join round to eliminate symmetric matches, while intermediate results must retain symmetric terms to ensure that each join round covers all possible matches.

The time complexity of Enhanced Secure Join is dominated by the oblivious compaction and injectivity verification components. For each join operation, the complexity is $O(|R| \log |R| + b \cdot |V(Q)|)$, where $|R|$ is the size of the input relation, b is the hybrid upper bound, and $|V(Q)|$ is the number of query vertices.

The $O(|R| \log |R|)$ comes from the sorting in oblivious compaction, while the $O(b \cdot |V(Q)|)$ arises from injectivity verification on the compacted results. Since b is typically much smaller than the Cartesian product size and $|V(Q)|$ is treated as a constant, the overall complexity is effectively $O(|R| \log |R|)$ per join operation.