Relational Algorithms for Top-k Query Evaluation

Anonymous Author(s)

ABSTRACT

The evaluation of top-k conjunctive queries, a staple in business analysis, often requires evaluating the conjunctive query prior to filtering the top-k results, leading to a significant computational overhead within Database Management Systems (DBMSs). While efficient algorithms have been proposed, their integration into DBMSs remains arduous. We introduce relational algorithms, a paradigm where each algorithmic step is expressed by a relational operator. This allows the algorithm to be represented as a set of SQL queries, enabling easy deployment across different systems that support SQL. We introduce two novel relational algorithms, **level-k** and **product-k**, specifically designed for evaluating top-k conjunctive queries and demonstrate that **level-k** achieves optimal running time for top-k free-connex queries. Furthermore, these algorithms enable easy translation into an oblivious algorithm for secure query evaluations. The presented algorithms are not only theoretically optimal but also exhibit eminent efficiency in practice. The experiment results show significant improvements, with our rewritten SQL outperforming the baseline by up to 6 orders of magnitude. Moreover, our secure implementations not only achieve substantial speedup compared to the baseline with secure guarantees but even surpass those baselines that have no secure guarantees.

ACM Reference Format:

Anonymous Author(s). 2024. Relational Algorithms for Top-k Query Evaluation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In real-world scenarios, exhaustive query results can often prove superfluous, with users generally seeking only the "best" or "most interesting" results. For instance, in Wikidata's SPARQL query service [60], more than 45.3% of queries contain a "LIMIT" clause to return only a small part of the full results [15]. The following also gives an example of a top-k conjunctive query[1].

Example 1.1. Consider a scenario where the government aims to organize drug trials, which involves a trilateral collaboration among hospitals, medical labs, and drug companies. Only hospitals and medical labs situated in the same city are eligible to collaborate. The government has evaluated and assigned scores to potential hospitals and medical labs to facilitate the selection process. These scores are cataloged in two relations: H(id, city, score) for hospitals and L(id, city, score) for labs. Additionally, each hospital has independently assessed and scored potential drug companies for collaboration, which is captured in relation D(hid, did, score).

Figure 1 gives a running instance of the database, and the following SQL query retrieves the combinations with the top 5 highest total scores for collaboration:

```
SELECT H.id, L.id, D.did
FROM H, L, D
WHERE H.city = L.city and H.hid = D.hid
ORDER BY H.score+L.score+D.score DESC
LIMIT 5;
```

The standard method for evaluating a top-k query involves evaluating the conjunctive query first, then sorting the results by annotation and outputting the top-k results. This approach is used by most of the commercial DBMSs[2] and has a cost of $O(N + |\mathcal{J}| \log |\mathcal{J}|)$, where N represents the input size of the database and $|\mathcal{J}|$ represents the output size of the conjunctive query and is typically much larger than k. The complexity can be further reduced by pushing down the order-by operator to the base table or intermediate results. However, such approaches may not always be feasible, and the worst-case complexity remains unchanged. In recent years, novel methods [15, 24, 58] have been proposed to approach the optimal complexity of this problem. However, the practical implementation of these methods in database systems remains challenging due to the customized operations and data structures they need.

On the other hand, with the rapid growth of data volumes, it becomes harder for users to manage and process all this information. Cloud services have stepped up as an effective solution, offering a cost-friendly and reliable method for data storage, computation, and transmission. This paradigm shift towards cloud computing has given rise to the trend of *data outsourcing*. However, there are hurdles to overcome, especially around privacy. Businesses, like hospitals in Example 1.1 that handle sensitive information, are concerned about information leakage.

A solution to this problem is to have cloud services keep only encrypted user data, leading to the creation of *encrypted databases* [11, 37, 47, 56, 66]. The method of *secure multi-party computation* (MPC) instantiates this concept. Leveraging multiple non-colluding servers, MPC breaks down and encrypts data into different parts called *shares*. Through a collaborative execution of a predetermined protocol, the servers perform computations on the shares, ensuring the theoretical protection of data during processing and assuring the authenticity of the outcomes.

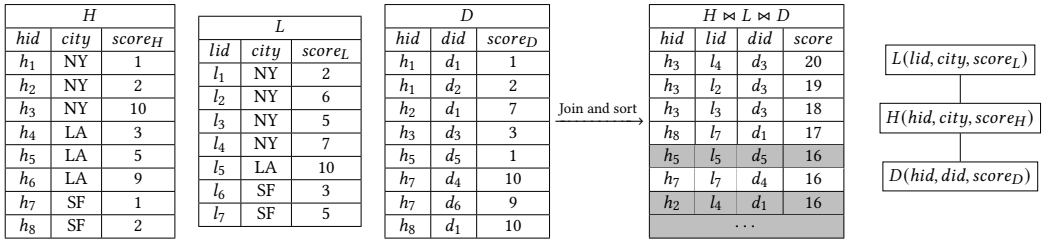
Yet, transitioning from an algorithm that computes over plaintext to one that fits within the MPC model is not straightforward. Although there are general solutions to securely perform operations (such as addition and multiplication [14, 30, 52, 68]) over shares, the

[1]We often use the term 'top-k query' as a shorthand representation for 'top-k conjunctive query' in the rest of the paper.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[2]See the source code from DuckDB [1] and PostgreSQL [2].



(a) Running example **(b) Join tree**

Figure 1: A running example and the join tree of Example 1.1

way data is accessed and changed in memory, known as the *access pattern*, must also be protected too [26]. This requires algorithms to be *oblivious*, i.e., the access pattern should be independent of the input. Techniques such as ORAM can render non-oblivious algorithms oblivious but often entail considerable overhead. The resultant systems are sometimes more than 1000 times slower than plaintext processing [49], due to both the overheads of computing over shares and making algorithms oblivious, which is a massive impediment that hinders practical application.

To understand why this huge gap exists, let's consider making the (natural) join operator oblivious. When joining two tables of size N, the complexity inevitably escalates to $O(N^2)$. Anything less would rule out the worst-case input, making the algorithm non-oblivious. Consequently, the naive algorithm that joins c tables one by one would result in a running time $O(N^c)$. In contrast, an algorithm for plaintext can have the advantage that the join conditions typically filter out many intermediate results. Some recent studies [9, 12, 36] have proposed a relaxation of security prerequisites, assuming that revealing the final result sizes might not compromise safety. While the validity of this proposition awaits confirmation, preliminary findings suggest that even with such relaxed constraints, the query time remains far from ideal. For example, the state-of-the-art secure query engine [36] takes 2000x more time than PostgreSQL on TPC-H Query 4 with 2^{16} tuples.

Top-k queries may pave the way for overcoming these hurdles. First, since k is typically far smaller than the worst-case output size $O(n^c)$, the theoretical minimum computation cost is reduced significantly from $O(n^c)$ to $\tilde{O}(n+k)$[3]. Additionally, k is a public parameter and can be disclosed during the execution, avoiding costly dummy tuple padding while still preserving private information.

1.1 Our Contribution

In this paper, we study top-k queries, especially focusing on designing relational algorithms with optimal guarantees. Intuitively, a relational algorithm is one in which all operations in the algorithm can be expressed by relational algebra and its extensions. Section 3 summarizes the relational operators used in the paper.

Our primary contribution is the introduction of **level-k** (Section 4.1), a relational algorithm that evaluates top-k binary join queries in time $O(N + k \log k)$. The achieved complexity is optimal, demanding at least $O(N)$ time for input reading and $O(k \log k)$ for output sorting[4]. We observe that **level-k** evaluates a top-k query in $\log k$ rounds, which might introduce an additional constant cost in practice. To address this limitation, we also present **product-k** (Section 4.1) — a variant of **level-k** that operates in $O(1)$ round. Although its worst-case running time is $O(N + k^2)$ (assuming $|\mathcal{J}| \geq k^2$), its implementation is more straightforward, and its practical performance could potentially surpass that of **level-k** due to a smaller hidden constant.

Our methodologies are not confined to binary join queries. We successfully adapt **level-k** and **product-k** to support full acyclic conjunctive queries and *free-connex* queries (Section 4.2). For these queries, we achieve a near-linear running time for these queries concerning the input size N and the final count k. Our approach incorporates a sequence of reductions, with each step diminishing the query size by 1. Once the query is reduced to a single table, it exactly stores the top-k query results. We prove that for any top-k free-connex queries, our method can solve them in optimal time $O(N + k \log k)$. Based on this, we further achieve $O(N^w + k \log k)$ running time for general conjunctive queries that are not free-connex (Section 4.3) by equipping **level-k** with Generalized Hypertree Decomposition (GHD) and Worst-Case Optimal Joins (WCOJ). The result further improves the complexity of the previous state-of-the-arts by a factor of $\log N$ (Section 4.4).

Furthermore, we demonstrate that (1) relational algorithms can be represented by a set of SQL statements, which allows us to deploy our novel algorithm on any database or data processing platform that supports SQL without modifying the kernel while ensuring optimal guarantees, and (2) relational algorithms can be easily turned into oblivious algorithms due to the natural of relational operators, making **level-k** and **product-k** to be oblivious (Section 5). We implement it within the three-server model, leading to time and communication complexities of $O(N \log N + k \log k)$. To the best of our knowledge, this is the pioneering algorithm for top-k queries under the MPC model.

In addition to the theoretical significance, our algorithms prove potent in practical settings. To illustrate this, we engage in experimental evaluations in Section 6 on both plaintext and MPC:

- We express both **level-k** and **product-k** by a set of SQL statements, where **level-k** can be expressed by $O(\log k)$ SQL statements and **product-k** can be expressed by $O(1)$ SQL statements. Evaluations of the original and transformed queries on commercial DBMSs like PostgreSQL and DuckDB reveal that our algorithms offer 3 to 6 orders of magnitude improvement to the current database engines, even without considering those data points that the original queries cannot finish in 8 hours.

[3] $\tilde{O}$ notation suppresses log factors
[4] Comparison sorts on average cannot surpass $O(k \log k)$.[21]

- By implementing **level-k** and **product-k** on the well-known ABY3 framework, we are able to support top-k queries in secure multi-party settings. The experiment results indicate a huge improvement, surpassing even the performance of native database engines operating on plaintext. These findings not only attest to our algorithm's practical utility but also suggest its potential to expand the range of applications in privacy-preserving query evaluations.

1.2 Related Work

Optimizing conjunctive query evaluation is a pivotal challenge in database research due to its potential to generate huge intermediate or final results, which is critical for performance. Significant efforts have been made to discover instance or worst-case optimal algorithms, not only for conjunctive queries [8, 55, 67] but also their integration with other operators, such as aggregations [7, 42], unions [17], comparisons [43, 44, 62], set differences [38], and updates [39, 61].

Top-k queries are another important class of queries. In fact, they consist of two operators, "ORDER BY" and "LIMIT", where the "ORDER BY" operator sorts the query results, while the "LIMIT" operator constrains the size of the output results. [41, 59] provide good surveys of the problem. Threshold Algorithm (TA) [28], J* [54], Rank-Join [40], LARAJ* [50], and a-FRPA [29] uses heuristic and cost models to reduce search space. However, none of these approaches can provide an optimality guarantee. [15] studies threshold queries, which limits the final output size without ranked requirement. [58] and [23, 24] simultaneously studied the problem of rank enumeration over conjunctive queries, which can be regarded as a generalized problem of top-k queries. They imply $\tilde{O}(N+k)$ algorithms for top-k queries. When k is known in advance, we can achieve better complexity than using rank enumeration for solving top-k queries. A detailed analysis is given in Section 4.4. Based on Rank-Join [40], Li et al. [46] extended relational algebra and added new physical operators to implement top-k operators inside databases. They also proposed optimization techniques to push down sorting and limit operators on query plans. Still, those techniques rely on the special structure of the ranking function and cannot provide optimality guarantees. We adopt a similar setting as [46], while we (1) avoid introducing new operators inside databases but implement our techniques by query rewrite, which renders our approach easily implemented in different databases, and (2) our rewrite plan can be proved optimal. Nevertheless, all these previous works are not oblivious and cannot be made oblivious easily, while our approach is oblivious. Another line of studies [19] tries to optimize top-k queries on the external memory model, where the entire database or even K tuples cannot be fitted into the memory. The setting departs from the setting in this paper, where we focus on the RAM model such that the database can fit into the memory.

The aforementioned methods operate on plaintext without providing any security or privacy guarantee. Secure computation refers to a set of techniques employed to perform computations on data while ensuring the confidentiality of the data. At the heart of secure computation lies the concept of oblivious algorithms, which exhibit workflow and access patterns during processing that are independent of the input data. Krastnikov et al. [45] propose a general oblivious algorithm for binary (equi-)joins. This algorithm achieves a near-optimal complexity of $O(N\log^2 N + M\log M)$, where N and M denote the input and output sizes, respectively. Secure multi-party computation (MPC), which enables multiple parties to jointly compute a function without disclosing any participant's private input except the function output, was initially introduced in Yao's pioneering paper [68]. SMCQL [11] is the first query processing engine in the two-party semi-honest MPC model. Since then, several research efforts [47, 53, 56] have been dedicated to enhancing the performance of MPC protocols. The most advanced algorithms for query processing currently available are two concurrent works [9, 36], which offer solutions with complexities of $O(N\log^2 N + M)$ and $O(N\log N + M\log M)$, respectively. The secure top-k queries [51, 70] have been explored using homomorphic encryption techniques. While homomorphic encryption is theoretically intriguing, it is computationally expensive and often impractical for real-world applications [49]. Another closely related topic is the secure k-nearest neighbor (kNN) search [18, 65, 69], which can be viewed as an application of top-k queries from a single table.

1.3 Outlines

Section 2 formally defines the top-k conjunctive queries and provides a lower bound for the problem. In Section 3, we introduce relational algorithms and all relational operators that form the basic building block of our work. Section 4.1 presents our algorithms, **product-k** and **level-k**. They are further extended to support free-connex queries (Section 4.2) and general conjunctive queries (Section 4.3). Section 5 studies how to apply our algorithm under the MPC model. In Section 6, we present our system design and the experimental evaluation. At last, we conclude the paper in Section 7. Due to the space constraints, all proofs are given in the technical report [3].

2 PRELIMINARY

2.1 Top-k Conjunctive Queries

Conjunctive Queries. In this work, we focus on *conjunctive queries (CQs)* of the following form:

$$Q = \pi_{\boldsymbol{O}} \left(R_1(\boldsymbol{E}_1) \bowtie R_2(\boldsymbol{E}_2) \bowtie \cdots \bowtie R_c(\boldsymbol{E}_c)\right),$$

where $\boldsymbol{R} = \{R_1, \cdots, R_c\}$ is the set of all relations, and each $R_i(\boldsymbol{E}_i)$ is a relation with a set of attributes $\boldsymbol{E}_i$, for $i = 1, 2 \ldots, c$. For simplicity, suppose a relation R_i appears twice in the query (with different attribute renamings), then we consider them as two identical copies of R_i. $\boldsymbol{E} = \boldsymbol{E}_1 \cup \boldsymbol{E}_2 \cup \cdots \cup \boldsymbol{E}_c$ is the set of all attributes in the query, and $\boldsymbol{O} \subseteq \boldsymbol{E}$ is the set of *output attributes*. Let $\mathcal{J} = R_1(\boldsymbol{E}_1) \bowtie R_2(\boldsymbol{E}_2) \bowtie \cdots \bowtie R_c(\boldsymbol{E}_c)$. If $\boldsymbol{O} = \boldsymbol{E}$, such a query ($Q = \mathcal{J}$) is known as a *full join query*; otherwise ($Q = \pi_{\boldsymbol{O}}\mathcal{J}$), it is said to be a *join-project query*.

We adopt the standard RAM model of computation and measure the running time in terms of data complexity, i.e., the query size $|Q|$ is considered a constant. The asymptotically optimal running time for evaluating any query is $\tilde{O}(N+M)$, where $N = \sum_{i=1}^{n} |R_i|$ is the input size and $M = |Q|$ is the output size. Note that for the top-k query, the output size $M = k$.

Annotated Relations. We follow the same terminology of annotated relations from recent works [7, 42]. Let $(\mathcal{S}, \oplus, \otimes)$ be a communicative semiring, where $\mathcal{S}$ is the ground set and $\oplus$ and $\otimes$ are its "addition" and "multiplication" operators. Given such a semiring, for any tuple $t \in R_i$, we assign it with an annotation $v_i(t) \in \mathcal{S}$.

For a full join query $\mathcal{J}$, the annotation for any $t \in \mathcal{J}$ is

$$v(t) = \bigotimes_{R_i(\boldsymbol{E}_i) \in \mathcal{Q}} v_i(\pi_{\boldsymbol{E}_i} t).$$

For a join-project query $\mathcal{Q} = \pi_{\boldsymbol{O}} \mathcal{J}$ over the full join query $\mathcal{J}$, assume each tuple $t' \in \mathcal{J}$ has its annotation $v'(t')$, the annotation for any $t \in \mathcal{Q}$ is

$$v(t) = \bigoplus_{\forall t' \in \mathcal{J}, \pi_{\boldsymbol{O}} t' = t} v'(t').$$

In addition, we require the semiring to be a *selective dioid* [31, 58]. Selective dioids are semirings with an ordering property, where the $\oplus$ satisfies either $x \oplus y = x$ or $x \oplus y = y$. For example, the well-known tropical semiring $(\mathbb{N}, \max, +)$ is a selective dioid as $x \max y$ always returns x or y. From a selective dioid, we can further define an order $\preceq$ such that if $x \oplus y = y$, then $x \preceq y$. The order $\preceq$ satisfies

(1) If $x \preceq y$, then $x \oplus z \preceq y \oplus z$;
(2) If $x \preceq y$ and $0 \preceq z$, then $x \otimes z \preceq y \otimes z$ and $z \otimes x \preceq z \otimes y$, where 0 is the $\oplus$-identity element.

We also define $x \succeq y$ if $y \preceq x$; define $x \succ y$ if $y \preceq x$ and $y \neq x$.

Top-k Conjunctive Queries. Top-k query processing is an important building block in database systems as it can reduce query cost when only the "best" or "most interesting" results are needed instead of the full output. Given a conjunctive query $\mathcal{Q}$ over annotated relations with a selective dioid $(\mathcal{S}, \oplus, \otimes)$, the annotation of each output tuple, commonly a real or an integer, is the annotation $v(t)$. The top-k conjunctive query returns the k tuples with top-k annotations. We assume all annotation $v(t)$ are distinct; otherwise, the tie can be broken by enforcing an additional lexicographic order on the output attributes [58].

We define its output $\mathcal{T} = \lambda_k \tau_{\preceq}(t_1, t_2, \cdots, t_k) \subseteq \mathcal{Q}$, such that:

(1) $\forall 1 \leq i < j \leq k, v(t_j) \preceq v(t_i)$;
(2) $\forall t \in (\mathcal{Q} - \mathcal{T}), v(t) \preceq v(t_k)$.

Here, $\lambda_k(\mathcal{Q})$ limits the output to the first k outputs of $\mathcal{Q}$, $\tau_{\preceq}(\mathcal{Q})$ sorts $\mathcal{Q}$ on v with the order $\preceq$. In the standard SQL statement, the top-k query corresponds to the "ORDER BY LIMIT" clause, as

```
SELECT O, ⊕(v1(x1) ⊗ ··· ⊗ vn(xn))
FROM R1(x1) NATURAL JOIN ··· NATURAL JOIN Rn(xn)
GROUP BY O
ORDER BY ⊕(v1(x1) ⊗ ··· ⊗ vn(xn)) DESC LIMIT k;
```

Note that this clause also requires the output to be sorted by the annotations with order $\preceq$. In our definition, we do not specify any requirement for the output order. We could sort the output if it needs to be ordered, which incurs cost $O(k \log k)$.

2.2 Classification of CQs

To help present our results, we introduce some commonly used terminologies and important classes of CQs.

Acyclic CQ [13, 27] There are many equivalent definitions for acyclicity, and we adopt the one based on *join tree*. A CQ $\mathcal{Q}$ is acyclic if there exists a tree $\mathcal{T}$ satisfying the following properties: (1) the set of nodes in $\mathcal{T}$ is a one-to-one mapping to the set of relations in $\mathcal{Q}$; (2) for each attribute E, all nodes of $\mathcal{T}$ containing E form a connected subtree of $\mathcal{T}$.

Free-connex CQ [10] A CQ $\mathcal{Q}$ is *free-connex* if a join tree $\mathcal{T}$ exists whereby a subtree $\mathcal{T}_c$ that contains the root node has $\boldsymbol{E}_c$ as the set of all attributes presented in $\mathcal{T}_c$, with (1) $\boldsymbol{O} \subseteq \boldsymbol{E}_c$, and (2) for any non-root node relation $R(\boldsymbol{E})$ on $\mathcal{T}_c$, having $R_p(\boldsymbol{E}_p)$ as its parent node, $\boldsymbol{E} \cap \boldsymbol{E}_p \subseteq \boldsymbol{O}$. Such $\mathcal{T}$ is recognized as a free-connex join tree of $\mathcal{Q}$.

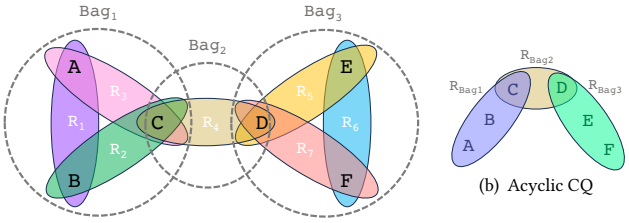
Both acyclic and free-connex characteristics bear significance in analyzing the hardness of a CQ. In fact, acyclic CQ represents the complete class of CQ that can be evaluated optimally in $O(N + M)$ time if the CQ is full [10, 67]. At the same time, free-connex is the complete class of CQ evaluated in $O(N + M)$ time, given that the query is non-full [10]. The hardness results are obtained by well-known conjectures, including Boolean Matrix Multiplication conjecture[5] and HyperClique conjecture[6]. These can be further extended to top-k queries, for example:

Theorem 2.1. *There is no algorithm that can evaluate the top-k query over CQ $\pi_{A,C} R(A, B) \bowtie S(B, C)$ in $\tilde{O}(N + k)$ time unless BMM conjecture fails.*

Generalized Hypertree Decomposition (GHD) [8, 34] is a powerful tool for query processing. It efficiently transforms cyclic CQs into acyclic forms, allowing for effective evaluation of non-acyclic queries. The GHD method involves grouping relations into "bags" and organizing these bags into a join tree, denoted as T. This process unfolds in two distinct phases. Initially, for each bag Bag, we compute the full join query $Q_{\text{Bag}} := \bowtie_{R \in \text{Bag}} R$ by using WCOJ [55]. The results from these queries create a special relation, R_{Bag}, with a worst-case size of N^w. Here, the treewidth w is a crucial parameter in tree decomposition, representing the worst-case output size of Q_{Bag} for each bag. Notably, a single relation on T has a weight of 1, suggesting that the treewidth for any acyclic CQ is also 1. After computing all Q_{Bag}, we replace all bags on T with R_{Bag}, forming a new tree T' that represents an acyclic CQ, $\hat{Q}$. The treewidth of T is the maximum treewidth among all bags, i.e., the maximum input size for all relations on T'. The second phase is evaluating this acyclic CQ. Using the Yannakakis algorithm [67], we can perform this evaluation in $O(N' + M)$ time, where $N' = N^w$ represents the current input size on T'. Among different GHDs, those with lower treewidth have better worst-case running times.

Finding the optimal GHD with the smallest treewidth is a complex task; even determining whether a GHD exists with treewidth below a certain constant is NP-complete [33]. Moreover, the treewidth can be further optimized by employing multiple GHDs [8]. In this study, we assume the optimal GHD or GHDs are provided beforehand, as we consider constant query size, and our objective is to evaluate top-k queries efficiently using the given GHD or GHDs.

[5]The Boolean Matrix Multiplication (BMM) conjecture [57] states: Given two Boolean matrices of size $n \times n$, no algorithm can compute their multiplication in $\tilde{O}(n^2)$ time.
[6]The HyperClique (HC) conjecture [48] states: given a k-uniform hypergraph (for $k \geq 3$), no algorithm can determine whether a hyperclique of size $k + 1$ exists or not in $O(m)$ time.



(a) Generalized Hypertree Decomposition (b) Acyclic CQ

Figure 2: An Example of GHD and its acyclic CQ

Example 2.2. See Figure 2(a) as an example of GHD on a natural join of 7 relations: $R_1(A,B)$, $R_2(B,C)$, $R_3(C,A)$, $R_4(C,D)$, $R_5(D,E)$, $R_6(E,F)$ and $R_7(F,D)$. There are three bags in the decomposition:

$$R_{Bag_1} \leftarrow R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,A);$$

$$R_{Bag_2} \leftarrow R_4(C,D);$$

$$R_{Bag_3} \leftarrow R_5(D,E) \bowtie R_6(E,F) \bowtie R_7(F,D).$$

After performing two triangle joins (R_{Bag_1} and R_{Bag_3}) we get an acyclic join (line-3 join) as Figure 2(b), with treewidth 1.5. The tree can be evaluated in a total time of $O(N^{1.5}+M)$.

3 RELATIONAL ALGORITHMS

In recent years, substantial efforts have been dedicated to enhancing efficiencies in query processing. Notwithstanding, a significant portion of these works verify their efficiency by implementing new methods in standalone research systems or specific products, thus impeding their implementation and validation within authentic environments. Recently, some attempts [32, 38, 63] based on query rewriting have emerged. By rewriting the original SQL into optimized SQL that follows their proposed algorithms, this approach facilitates the straightforward deployment of advanced algorithms with complexity guarantee, across different databases or data processing systems that support SQL.

However, not all algorithms can be directly translated by SQL statements. To further this concept, we introduce **relational algorithms**, representing a category of algorithms that can be fully articulated through relational algebra [6] and its extension [46]. These operators can all be directly expressed as SQL expressions and, when executed within a database, come with running time guarantees. Table 1 summarizes the relational operators used in our study, as well as some expansions that have been studied before, but are yet to be discussed as relational operators, including:

Top-k($\lambda_k^{\preceq}(R)$): given a table (or query result) R, it returns the k tuples of R with largest annotations. It simplifies $\lambda_k \tau_{\preceq}(R)$ without requiring the final output to be sorted. The standard approach for the top-k operator in databases would require first sorting the table R, then outputting the first k results, making the running time $O(|R|\log|R|+k)$. However, it can be further improved to $O(|R|)$ under the RAM model by using the well-known quick select algorithm [20] to find the k-th largest annotation v_k. The quick select algorithm is very similar to the quick sort algorithm. Each step takes a pivot and partitions the input array into two parts according to the pivot. The difference is that finding v_k requires the recursion only going into one part, making the total cost of the quick select algorithm $O(|R|)$. After finding v_k, one can simply scan all tuples in R and keep only the tuples with annotations larger or equal to v_k as a result. Therefore, the total cost is $O(|R|)$.

Row-number($\rho_{\boldsymbol{E}}^{\preceq}(R)$): Given a table (or query result) R, it assigns a unique, sequential integer into column id to each row. The rows are partitioned by $\boldsymbol{E}$ and ordered according to annotation v. The integer is assigned incrementally from 1 within each group. When $\boldsymbol{E}=\emptyset$, the entire R is considered a single group, and id is allocated in accordance with the overall ordering determined by annotation v. The row-number operator is not derived from relational algebra but originates from the SQL ANSI 2003 standard [25].

In addition, we can derive the following principle:

Theorem 3.1. (quasi-commutative principle). *For any relation $R(F)$ and $E \subset F$, if the annotations of $R(F)$ are all distinct, then*

$$\pi_E^{\oplus}\left(\lambda_k^{\preceq}(R)\right) \subseteq \lambda_k^{\preceq}\left(\pi_E^{\oplus}(R)\right).$$

As a consequence, any relational algorithm can be converted into SQL and submitted directly for execution within a database, without the necessity to modify the database kernel or create standalone research systems.

4 THE TOP-K ALGORITHM

4.1 Top-k Binary Join

We start with the top-k binary join, the simplest non-trivial top-k problem. Let R and S be two relations and $|R|=|S|=O(N)$. Our target is to compute $T=\lambda_k^{\preceq}(R\bowtie S)$. Without loss of generality, let R have two attributes A and B and S have B and C, then T contains three attributes (A,B,C) and the join condition of $R\bowtie S$ is on their common attribute B. The standard approach considers the join and top-k as separate operators. By calculating the join $T'=R\bowtie S$ first, then calculating the top-k on single table T', such an approach would take at least $O(|T'|+k)$ time, where $|T'|=N^2$ in the worst case, which can be significantly larger than N or k.

Tziavelis et al. [58] introduced an innovative algorithm designed to evaluate top-k binary join queries in a time complexity of $O(N+k\log k)$. The authors ingeniously transformed the join problem into a special k-longest path problem by constructing a directed bipartite graph from instances of relations R and S. Precisely, vertices v_r on the right-hand side were mapped one-to-one to each tuple in R or S, while vertices v_l on the left-hand side were mapped to each value of attribute B. Additionally, two unique vertices, denoted as source and sink, were introduced on the left-hand side. Connections were established from source to all vertices in R and from all vertices in S to sink with a length of 0. For every vertex in $t\in R$ on the right-hand side, a connection was made to $\pi_B t$ on the left-hand side, with a length equivalent to the annotation $v(t)$; similarly, for every vertex in $t\in S$ on the right-hand side, a connection was made from $\pi_B t$ to t, with a length equivalent to $v(t)$. Consequently, the top-k binary join results correspond to the top-k longest paths between source and sink. An illustrative example of the bipartite graph is provided in Figure 3. To identify the k-longest path problem, the authors employed a modified dynamic programming technique, diverging from maintaining a single maximum/minimum value,

Operator	SQL Query	Plaintext Complexity	Secure Complexity
Selection($\sigma_f(R)$)	`SELECT * FROM R WHERE f;`	$O(\|R\|)$	$O(\|R\|)$
Projection($\pi_{\boldsymbol{E}}(R)$)	`SELECT` $\boldsymbol{E}$ `FROM R;`	$O(\|R\|)$	$O(\|R\|)$
Group-By($\xi_{\boldsymbol{E}}(R)$)	`SELECT` $\boldsymbol{E}$ `FROM R GROUP BY` $\boldsymbol{E}$`;`	$O(\|R\|)$	$O(\|R\|)$
Order-By($\tau_{\preceq}(R)$)	`SELECT * FROM R ORDER BY` v `DESC;`	$O(\|R\|\log(\|R\|))$	$O(\|R\|\log(\|R\|))$
Limit(λ_k)	`SELECT * FROM R LIMIT k;`	$O(k)$	$O(k)$
Join($R_1 \bowtie R_2$)	`SELECT *,` $R_1.v \otimes R_2.v$ `AS` v `FROM` R_1 `NATURAL JOIN` R_2`;`	$O(\|R_1\| + \|R_2\| + \|R_1 \bowtie R_2\|)$	$O(\|R_1\| \cdot \|R_2\|)$
Union($R_1 \cup R_2$)	`SELECT * FROM` R_1 `UNION` R_2	$O(\|R_1\| + \|R_2\|)$	$O(\|R_1\| + \|R_2\|)$
SemiJoin($R_1 \ltimes R_2$)	`SELECT * FROM` R_1 `WHERE EXISTS (SELECT 1 FROM` R_2 `WHERE` $R_2.key = R_1.key$`);`	$O(\|R_1\| + \|R_2\|)$	$O(\|R_1\|\log(\|R_1\|) + \|R_2\|\log(\|R_2\|))$
Group-By Aggregation($\pi^{\oplus}_{\boldsymbol{E}}(R)$)	`SELECT` $\boldsymbol{E}$`,` $\oplus(v)$ `AS` v `FROM` R `GROUP BY` $\boldsymbol{E}$`;`	$O(\|R\|)$	$O(\|R\|\log(\|R\|))$
Top-k($\lambda^{\preceq}_k(R)$)	`SELECT * FROM R ORDER BY` v `DESC LIMIT k;`	$O(\|R\|)$	$O(\|R\|)$
Row-Number ($\rho^{\preceq}_{\boldsymbol{E}}(R)$)	`SELECT *, row_number() OVER (PARTITION BY` $\boldsymbol{E}$ `ORDER BY` v `DESC) AS id FROM` R`;`	$O(\|R\|\log(\|R\|))$	$O(\|R\|\log(\|R\|))$

Table 1: Summary of relation operators, where v represents the annotation

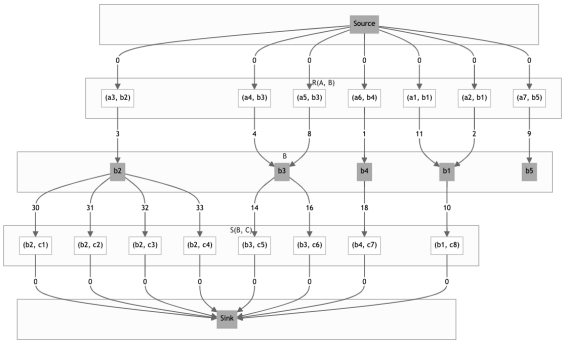


Figure 3: The transformed bipartite graph based on the running instance. Vertices in grey are the vertices on the left-hand side, and white are on the right-hand side.

preserving at most k values for each entry to retrieve k results expediently.

Despite the above method offering an optimal running time guarantee, transforming the problem into the shortest path problem is non-trivial and hard to implement inside a database system without modifying the kernel. Furthermore, its dependency on specific data structures, such as heaps, to assure runtime may hamper practical efficiency. To remedy these deficiencies, we introduce a relational algorithm, **level-k**, which is based on the following observation:

Proposition 4.1. *For any $(a,b) \in R$, let $(b,c_1),\ldots,(b,c_l)$ be tuples in S that can join with (a,b), and $v(b,c_1) \succ v(b,c_2) \succ \cdots \succ v(b,c_l)$. $(a,b,c_i) \notin T$ implies $(a,b,c_j) \notin T$ for any $j > i$, as $v(a,b,c_i) = v(a,b) \otimes v(b,c_i) \succ v(a,b) \otimes v(b,c_j) = v(a,b,c_j)$.*

In simpler terms, if we have found k numbers larger than $v(a,b,c_i)$, it becomes unnecessary to evaluate any $j \geq i$, as there will invariably be at least k numbers surpassing $v(a,b,c_j)$ as well. We add a column ID to S indicating its relative order (i.e., (b,c_i) gets $ID \leftarrow i$).

Leveraging the above proposition, we introduce our algorithm **level-k**. The first step of the algorithm aims to eliminate unnecessary tuples from S if $|S| > k$. Given that at most k distinct S tuples can present in the final query results, this is a prudent step. Meanwhile, we calculate the ID utilizing the Row-number(ρ) function. By the quasi-commutative principle, we deduce that

$$\pi^{\oplus}_{B,C}\left(\lambda^{\preceq}_k(R \bowtie S)\right) \subseteq \lambda^{\preceq}_k\left(\pi^{\oplus}_{B,C}(R \bowtie S)\right).$$

Here, the left-hand side furnishes tuples in S that contribute to the top-k binary join result, which is a subset of the right side denote as S'. Therefore, replacing S with S' does not affect the correctness of the query, i.e., $\lambda^{\preceq}_k(R \bowtie S) = \lambda^{\preceq}_k(R \bowtie S')$, while the size of S' is at most k.

The algorithm is then divided into $\log(k)$ rounds. Here, we use 2 as the base in the following discussion. In the first round, instead of joining each $(a,b) \in R$ with all possible $(b,c_i) \in S$ pairs, we only choose the top two, (b,c_1) and (b,c_2), to join. From the join result, labeled T_1, we take the first top-k tuples and use them to filter out all tuples for the next round. If (a,b,c_2) is not in the k results from the first round, then (a,b,c_i) for any $i \geq 2$ will not be in the final top-k query results, based on the proposition. So, we only need to filter out all (a,b) so that (a,b,c_2) is in the k results as a candidate for the second round.

Example 4.2. Figure 4 shows an instance of the top-k query $\lambda^{\preceq}_k(R(A,B) \bowtie S(B,C))$. For the first round of computation, **level-k** will first compute $S \leftarrow \rho^{\preceq}_B \lambda^{\preceq}_k(S \bowtie \pi^{\oplus}_B(R))$. The result is marked with white in the $\rho^{\preceq}_B(S)$ table, and now only 5 tuples are left in S. After that, **level-k** selects all tuples $t \in S$ where $t.ID \leq 2$, and (b_2,c_3), (b_2,c_4), and (b_3,c_6) are returned as S_1. By computing $R_1 \bowtie S_1$, we get the result for T_1. It's important to note that (a_5,b_3,c_5) has a larger annotation than (a_4,b_3,c_6), but since (b_3,c_5) is not in the top-5 results of $\pi^{\oplus}_{B,C}((R(A,B) \bowtie S(B,C))$, we don't consider that record as it cannot appear in the final query results.

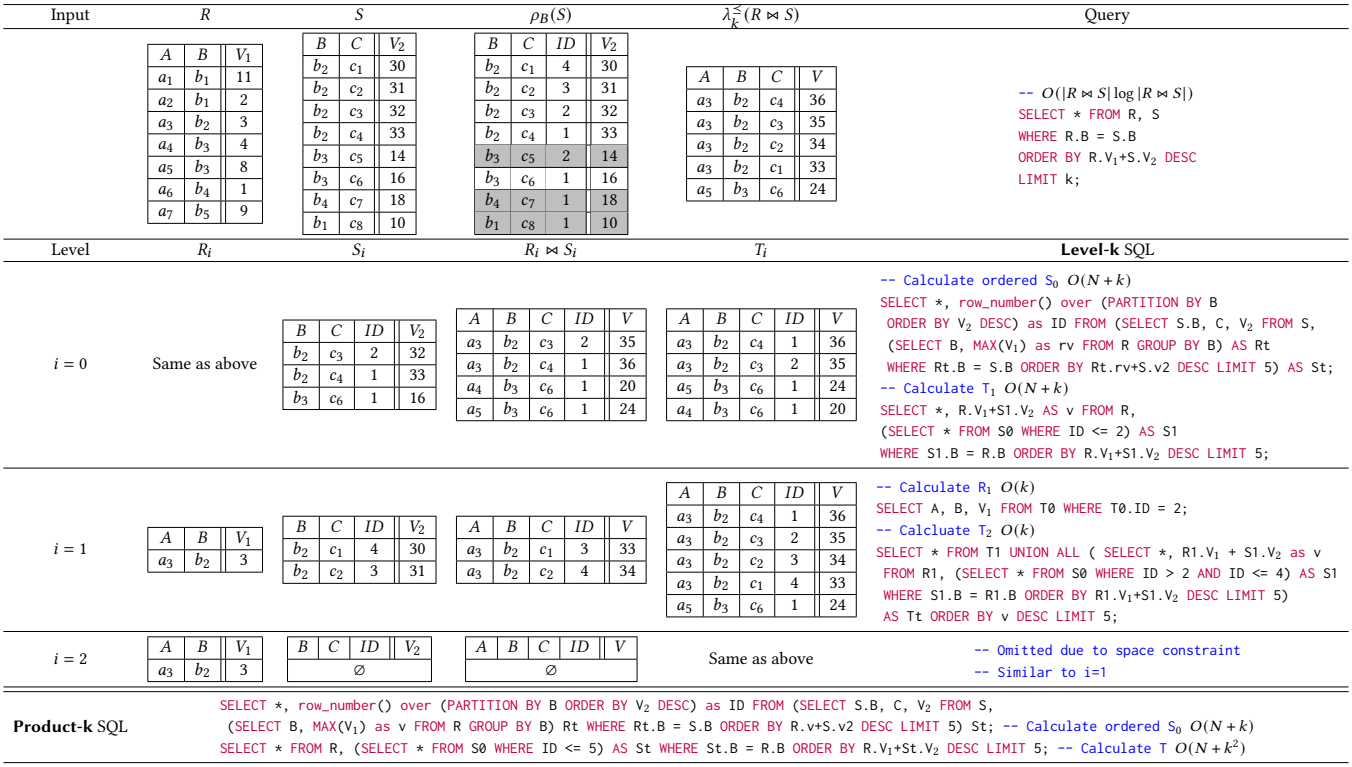


Figure 4: A running example of top-5 binary join query $R(A, B) \bowtie S(B, C)$.

After completing the first round of computation, we have at most $k/2$ candidates from R_1 for the second round. For each $(a, b) \in R_1$, we select the tuples (b, c_3) to (b, c_{2^2}) and compute T_2. Next, we union T_1 and T_2, filter out the top-k results, and choose all $R_2 \subseteq R_1$ so that for any $(a, b) \in R_2$, $(a, b, c_4) \in T_1 \cup T_2$.

Example 4.3. Moving forward with Example 4.2, in the second round of computation, since only one tuple (a_3, b_2, c_3) in T_1 has $ID = 2$, we can filter out other tuples from R_1, leaving just the tuple (a_3, b_2) in R_2. At the same time, we select tuples from S into S_2 where $B = b_2$ and $2 < ID \leq 4$, returning two tuples. Similarly, we compute $R_2 \bowtie S_2$, union the result with T_1, and retain the 5 tuples with the largest annotations. The tuple (a_4, b_3, c_6) is removed from the candidate results during this step.

The procedure unfolds recursively, as shown in Algorithm 1. In each iteration, the size of R_i is at most $k/2^i$. Once i reaches $\log k$, the size of R_i narrows down to 1, thereby enabling the computation to stop after $\log k$ iterations.

Example 4.4. Building upon Example 4.2, in the third round computation, given that (a_3, b_2, c_1) is the sole tuple with $ID = 4$, only (a_3, b_2) is retained in R_2. Nevertheless, no tuples in S satisfy the conditions $B = b_2$ and $4 < ID \leq 8$. Consequently, the results from the second round are preserved unchanged. As $2^3 > 5$, three computational rounds suffice for the algorithm to reach termination.

Theorem 4.5. *Algorithm 1 correctly returns $\lambda_k^{\preceq}(R \bowtie S)$ and runs in time $O(N + k \log k)$ with base $b = 2$.*

Algorithm 1: Level-k algorithm for top-k binary join

Input: Relations $R(A, B)$ and $S(B, C)$
Output: Top-k binary join result $T(A, B, C) = \lambda_k^{\preceq}(R \bowtie S)$

$S \leftarrow \rho_B^{\preceq} \lambda_k^{\preceq}((S \bowtie \pi_B^{\oplus}(R)))$; // Eliminate unnecessary tuples from S.
$S_0 \leftarrow \sigma_{ID \leq 2}(S)$; // Get the top-2 annots for all B.
$T_0 \leftarrow \lambda_k^{\preceq}(R \bowtie S_0)$; // First round join.
for $i \leftarrow 1$ **to** $\lceil \log \min(n, k) \rceil - 1$ **do**
  $R_i \leftarrow \pi_{A,B}(\sigma_{ID=2^i}(T_{i-1}))$; // Find candidates in R for the $(i+1)$-th round.
  $S_i \leftarrow \sigma_{2^i < ID \leq 2^{i+1}}(S)$; // Get tuples in S for the $(i+1)$-th round.
  $T_i \leftarrow \lambda_k^{\preceq}(T_{i-1} \cup (R_i \bowtie S_i))$; // Merge top-k results with the $(i+1)$-th round join.
$T \leftarrow \pi_{A,B,C}(T_{\lceil \log n \rceil - 1})$;
return T

Reducing the number of rounds. We demonstrated that **level-k** can operate optimally within time $O(N + k \log k)$. Although increasing the base may theoretically diminish running time, the algorithm still executes in $\log k$ rounds. In real-world scenarios, k is often relatively small, prompting a desire to decrease the round number for enhanced practical performance. To achieve this, we introduce **product-k**. After computing R_1 and T_0, we promptly select all (b, c_i) with $i \leq k$, executing the join thereafter. The algorithm is elaborated in Algorithm 2.

Algorithm 2: Product-k algorithm for top-k binary join

Input: Relations $R(A,B)$ and $S(B,C)$
Output: Top-k binary join result $T(A,B,C) = \lambda_k^{\preceq}(R \bowtie S)$

$S \leftarrow \rho_B^{\preceq}\lambda_k^{\preceq}(S \bowtie \pi_B^{\oplus}(R))$; // Eliminate unnecessary tuples from S.
$S_0 \leftarrow \sigma_{ID\le 1}(S)$; // Get the max annot. for all B. // Get the k possible tuples in R.
$T_0 \leftarrow \lambda_k^{\preceq}(R \bowtie S_0)$;
$R_1 \leftarrow \pi_{A,B}(T_0)$;
// Join R_1 and S to get the final results.
$S_1 \leftarrow \sigma_{ID\le k}(S)$;
$T \leftarrow \pi_{A,B,C}(\lambda_k^{\preceq}(R_1 \bowtie S_1))$;
return T

Product-k may be viewed as a specific instantiation of **level-k**, wherein we assign the base as k. The computations for S_0, T_0, and R_1 merely consume $O(N)$ time. Meanwhile, R_1 could have at most k tuples. For every $(a,b) \in R_1$, a maximum of k join tuples in S_1 may be present, leading the computation of T to take $O(k^2)$ time.

Theorem 4.6. *Algorithm 2 runs in time $O(N + k^2)$.*

SQL Rewrite. Since both **product-k** and **level-k** are relational algorithms, every step of the algorithms can be written into SQL queries, and the execution time of each query is bounded, as stated in Table 1 and Figure 4.

4.2 Top-k Free-connex CQs

The binary join algorithms can be further extended to accommodate a top-k free-connex Conjunctive Query $\lambda_k^{\preceq}(Q)$, where Q is a free-connex CQ. Let $\mathcal{T}$ be the free-connex join tree of Q. Throughout our algorithms, we employ a series of reduction procedures. For each reduction $Q(\boldsymbol{R}) \rightarrow Q'(\boldsymbol{R'})$, we ensure the following:

- Q' is still a free-connex CQ;
- $|\boldsymbol{R'}| = |\boldsymbol{R}| - 1$, and we can compute $\boldsymbol{R'}$ in $O(N + k\log k)$ (or $O(N + k^2)$) time;
- $\lambda_k^{\preceq}(Q') = \lambda_k^{\preceq}(Q)$.

Initialization. The initialization phase of our algorithm eliminates all non-output attributes from Q. Specifically, this step can be skipped when Q is already a full join query. Let $R(E)$ be any leaf node devoid of unique output attributes and $R_p(E_p)$ as its corresponding parent node. Subsequently, R can be removed by replace R_p with

$$R'_p := \pi^{\oplus}_{E_p}(R_p \bowtie R) = R_p \bowtie (\pi^{\oplus}_{E_p \cap E} R).$$

The query can be computed in an optimal $O(N)$ time complexity by pushing down the aggregation. For each $t \in R'_p$, its annotation $v(t)$ equals the sum of the original annotation of $t \in R_p$ and the maximal annotation from R that can join with t. The initial query Q is demonstrated to be equivalent to Q'.

$$\lambda_k^{\preceq}(Q') = \lambda_k^{\preceq}\left(\pi^{\oplus}_{\boldsymbol{O}}\left(\bowtie_{R' \in \boldsymbol{R} - \{R, R_p\}} R' \bowtie R'_p\right)\right)$$
$$= \lambda_k^{\preceq}\left(\pi^{\oplus}_{\boldsymbol{O}} \bowtie_{R' \in \boldsymbol{R}} R'\right) = \lambda_k^{\preceq}(Q)$$

We recursively remove all such R, subsequently obtaining a reduced query Q' on relation $\boldsymbol{R'}$. It is noteworthy that some relations $R(E) \in \boldsymbol{R'}$ may still have unique non-output attributes, and we need to remove them by replacing all such R with R' by

$$R' = \pi^{\oplus}_{E \cap \boldsymbol{O}} R. \tag{1}$$

Calculating R'_p or R takes only $O(N)$ cost. After the reduction, we can obtain a full join query Q' such that $\lambda_k^{\preceq}(Q') = (Q)$.

Reduction. For the reduced full join query Q, let $R(E)$ be the leaf relation and $R_p(E_p)$ be its parent. Invoking the quasi-commutative principle (Theorem 3.1), we have

$$\pi^{\oplus}_{E \cup E_p}\left(\lambda_k^{\preceq}(Q)\right) \subseteq \lambda_k^{\preceq}\left(\pi^{\oplus}_{E \cup E_p}(Q)\right).$$

This reveals that R and R_p can be replaced with the relation $T := \lambda_k^{\preceq}\left(\pi^{\oplus}_{E \cup E_p}(Q)\right)$ which size is bounded by k. A direct evaluation of the query above might lead to an $O(N^2)$ running time, given that $\pi^{\oplus}_{E \cup E_p}(Q)$ can yield up to $O(N^2)$ results. However, we establish that T can be computed in $O(N + k\log k)$ or $O(N + k^2)$ time:

Lemma 4.7. *Given a full join query Q, let $R(E)$ be the leaf relation and $R_p(E_p)$ be its parent, $T := \lambda_k^{\preceq}\left(\pi^{\oplus}_{E \cup E_p}(Q)\right)$ can be calculated in $O(N + k\log k)$ time by using **level-k** algorithm, or $O(N + k^2)$ time by using **product-k** algorithm.*

Based on Lemma 4.7, we can reduce $Q(\boldsymbol{R})$ to $Q'(\boldsymbol{R'})$ in $O(N + k\log k)$ or $O(N + k^2)$ time, where $Q' = \lambda_k^{\preceq}(\bowtie_{R' \in \boldsymbol{R'}} R')$ and $\boldsymbol{R'} = \boldsymbol{R} \cup \{T\} - \{R, R_p\}$. We repeat this process until only one relation is left in the entire query, and the relation stores the exact result of the top-k free-connex CQ.

Since we consider the data complexity and assume $|Q| = O(1)$, combining the above results, we can obtain the following theorem:

Theorem 4.8. *Top-k free-connex CQs can be solved in $O(N + k\log k)$ using **level-k** or $O(N + k^2)$ using **product-k**.*

4.3 Top-k CQs

As highlighted in Section 2, a non-acyclic query can be converted into an acyclic query or even a free-connex query by applying Generalized Hypertree Decomposition (GHD). Similar to the process used for full join queries, given a GHD with a treewidth w, each bag query Q_{Bag} is first evaluated in $O(N^w)$ time using the Worst-case Optimal Join algorithm. This step transforms each bag into a special relation, making the residual query acyclic. With this transformation, we can then directly apply either the **level-k** or the **product-k** algorithm for further processing. The procedure is summarized in Algorithm 3. Combining Theorem 3.1, we can show the following theorem for general conjunctive queries.

Theorem 4.9. *A top-k query Q can be evaluated in $O(N^w + k\log k)$ time using the **level-k** algorithm (or $O(N^w + k^2)$ using **product-k**), where w is the width of the optimal GHD/GHDs for Q.*

Algorithm 3: Evaluating top-k queries with GHD

Input: A CQ Q with a given GHD $\{\text{Bag}_1, \cdots, \text{Bag}_d\}$ on database instance D.
Output: Top-k result of $Q(D)$
// Compute the full join for each bag
forall $i \in [d]$ **do**
  $R_{\text{Bag}_i} \leftarrow \bowtie_{R \in \text{Bag}_i} R$;
// Q is equivalent to $\bowtie_{i\in[d]} R_{\text{Bag}_i}$.
// Evaluate the resulted acyclic query
$Q' := \lambda_k^{\preceq}(\bowtie_{i\in[d]} R_{\text{Bag}_i})$ using **product-k** or **level-k**;
return Q'

4.4 Top-k Queries v.s. Rank Enumeration

Rank enumeration [23, 24, 59] focuses on the sequential output of query results in a specific order. The formal definition is as follows:

Definition 4.10. Given a CQ Q over a database D and a ranking function, ranked enumeration outputs the query answers $Q(D)$ sequentially in ascending $\preceq$ order while ensuring no duplicates.

Rank enumeration can handle top-k queries by maintaining a counter during result enumeration. When this counter hits the value k, the enumeration stops, and the results up to that point constitute the top-k query results. However, rank enumeration requires one more capability beyond that of top-k queries: it must be able to enumerate the $(k+1)$-th result following the k-th result for arbitrary k. This requirement is not present in standard top-k queries. As a consequence, the algorithms cannot discard tuples that do not contribute to the final k results, and pre-processing the entire database is required. Additionally, each tuple should be stored in a heap-like structure to retrieve the subsequent tuple efficiently. These extra requirements cause rank enumeration to be less efficient for solving top-k queries, leading to a complexity of $O(N^w \log N + k \log N)$. Here, the extra $\log N$ factor in $N^w \log N$ is due to the extensive pre-processing across the entire database after pre-processing each bag in N^w time, while the additional $\log N$ factor in $k \log N$ arises from managing the heap-like structure that contains all N elements. Notably, rank enumeration is not a standard operator in the SQL standard and is not commonly utilized in well-known database products. In practical scenarios, the value of k is always predetermined. By capitalizing on this knowledge of k beforehand, we can enhance the efficiency of the methods proposed in [23, 24, 59], reducing the complexity by a factor of $\log N$.

5 SECURE TOP-K QUERY PROCESSING

Another advantage of the relational algorithm is data-independent, i.e., the algorithm is only decided by the query, not the data. Data-independent is a necessary condition for secure computation, which aims to protect the access pattern over different input instances. Our algorithms directly apply to an arbitrary secure model if there is a secure implementation of all relation operators listed in Table 1 under the model. As an example, we discuss how to instantiate our algorithms within the three-server honest majority model in this section. Despite simply replacing operators with their underlying secure implementations, we note that **level-k** involves repeatedly joining tables with similar structures for $\log k$ times and leveraging this property to save a $\log k$ factor. See Section 5.3 for details.

5.1 The Three-Server Model

In the three-server honest majority model (or the *three-server model* for brevity), there are three different types of roles: one or more data owners who hold the original data, three non-collude servers who are responsible for computation, and a client who submits a query to the servers and receives query results from them after computation. To protect the privacy of the data, the data owners send their data to the servers in a *secret shared* manner. The servers compute over the shared data by a *secure protocol*, which finally outputs the shares of the query result. Finally, the servers send the shares of outputs to the client, who reconstructs them to obtain the plaintext query result.

The three-server model is a special type of outsourcing model, where data owners outsource their data to the cloud (the three servers), which provides service for storage and computation. It disables the possibility that the cloud steals sensitive data from the data owners. Moreover, it enables computation over joint data from multiple data owners, which potentially makes data more valuable.

Secret sharing. The replicated secret-sharing scheme [52] is the most popular scheme under the three-server model, due to its simplicity and high efficiency. This could be the primary reason why the three-server model is more popular than the two-server model, as in the two-server model, computing over shares is complicated and of low efficiency.

In replicated secret-sharing, an ℓ-bit secret $v \in \{0,1\}^\ell$ is split into three strings with ℓ bits $\{v_i\}_{i=0}^2$, where each v_i is a uniformly random string in $\{0,1\}^\ell$, with the constraint that the logical XOR of them is equal to v. The i-th server holds v_i and $v_{(i+1) \bmod 3}$. Any two servers can reconstruct v jointly, while each server learns nothing about v. In this paper, unless specified, all the input and output of the functionalities or protocols are in the replicated secret-shared form.

Mohassel and Rindal [52] introduce how to efficiently perform basic operations (logical AND, OR; arithmetic addition, multiplication; comparison, etc.) securely over the replicated secret shares. Recent papers [35, 36] have shown how to perform relational operators securely over shared relations. The details are given later.

Cost model. Under the three-server model, both computation cost and communication cost between the servers are measured. In our theoretical analysis, we simply use "cost" to refer to any of them, as all of the protocols presented have the same complexity on the two costs.

If the maximum network latency between the three servers is significant, then the number of communication rounds should also be considered. All protocols in this paper have $O(\log(N+k))$ rounds, so the total latency overhead is negligible.

5.2 Relational Operators Protocols

This section introduces the efficient protocols for common relational operators under the three-server model, as summarized in

Table 1. Specifically, the protocol for the "Order-By" operator proposed in [35] has cost $O(N \log N)$ under the three-server model. Wang and Yi [64] have introduced Boolean circuits for the operators "Selection", "Projection", "Group-By (Aggregation)" and PK-FK join. By evaluating these circuits under the three-server model and replacing the underlying sorting circuit with the "Order-By" operator, their costs are $O(N)$, $O(N)$, $O(N \log N)$, and $O(N \log N)$, respectively. We simply compute the nested loop join for a general binary join without the PK-FK constraint, which incurs $O(N^2)$ cost.

In the PK-FK join $R \bowtie S$ where S is unique on the join key, one observation is that if R is already ordered by the join key, then there exists an $O(N)$ protocol to compute the join result: the initial step involves replacing all equivalent join keys in R, except the first one of them, with dummy elements to ensure that each join key in R is unique. Subsequently, a straightforward PK-PK join (as described in [53]) is executed such that the true tuple (without any changes to the dummy) receives the join tuple from S. The remaining tuples receive a value of 0. Finally, a segmented prefix-sum circuit (proposed in [64]) duplicates previous join tuples for subsequent tuples with identical join keys.

The secure "Top-k" operator adopts the idea from [35] that any comparison-based algorithm is oblivious after the input data is randomly shuffled. Thus, we can simply convert secure quicksort to secure top-k by changing the quicksort algorithm to the quick select algorithm, which has cost $O(N)$.

The secure "Row-Number" operator begins by grouping the relation based on the attribute $\boldsymbol{E}$. Equivalent tuples are then arranged in descending order according to their annotation. Subsequently, a segmented prefix-sum circuit is employed to compute the row number within each segment, where tuples sharing the same $\boldsymbol{E}$ form a segment. This operation incurs $O(N \log N)$ cost.

5.3 Secure Implementation of Level-k

So far, we have introduced the secure implementations of all the relational operators involved in the **product-k** algorithm, and the total cost of **product-k** under the three-server model is $O(N \log N + k^2)$. In this section, we will show that the **level-k** algorithm for top-k binary join is $O(N \log N + k \log k)$, so the total cost of **level-k** algorithm for any top-k free-connex conjunctive query.

In Algorithm 1, the join operators are binary join without limitation, which has a huge complexity ($O(N^2)$) in secure settings. Our goal is to simplify them to PK-FK joins so that the cost can be reduced to $O(N \log N)$. Our method is that instead of filtering S_i in each level, we assign attribute ID to R_i as well, where $ID = i$ represents the tuple that needs to join with the i-th largest tuple in S. We make the join as a PK-FK join, with join key $\{ID\} \cup \boldsymbol{E}$. Concretely, we copy each tuple in R_i 2^i times, and attach a new attribute ID from $2^i + 1$ to 2^{i+1}. In the expanded relation, the first $|R_i|$ tuples have $ID = 2^i + 1$, the second $|R_i|$ tuples have $ID = 2^i + 2$, and so on. Furthermore, we previously sort relation R by $\boldsymbol{E}$ in order, and the expanded relation is obviously ordered by ID then $\boldsymbol{E}$. Thus, the PK-FK join without sorting only takes $O(k)$ costs.

Then all operators in each level ($i \geq 1$) take $O(k)$ costs, and the first join at ($i = 0$) has $O(N \log N)$ cost, so the total cost is $O(N \log N + k \log k)$.

Algorithm 4: Ideal Functionality $\mathcal{F}_{\text{topk}}$

Input: Free-connex CQ; relations $[\![R_1(\boldsymbol{F_1}; V)]\!], \cdots, [\![R_k(\boldsymbol{F_k}; V)]\!]$; limit k
Output: Query result $[\![\mathcal{T}(\boldsymbol{O}; V)]\!]$

1 Recover R_i from $[\![R_i]\!]$ for $1 \leq i \leq k$;
2 $\mathcal{J} \leftarrow R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$; // Calculate the full join results.
3 $\mathcal{Q} \leftarrow \pi_{\boldsymbol{O}}(\mathcal{J})$; // Project on the output attributes.
4 $\mathcal{T} \leftarrow \lambda_k^{\preceq}(\mathcal{Q})$; // Find the top-k annotations.
5 Compute the secret share $[\![\mathcal{T}]\!]$ of $\mathcal{T}$;
6 **return** $[\![\mathcal{T}]\!]$

5.4 Security Guarantee

Our protocol is a sequential composition of existing semi-honest building blocks whose security has been established by prior work; all intermediate results are stored in secret-shared form, and the randomness is all independent. Thus, our protocol is secure against *semi-honest* adversary [16]. The adversary can potentially corrupt any subset of data owners, the client, and, at most, one server. All the corrupted parties are assumed to follow the protocol but attempt to gather additional information or deduce sensitive details from the information exchanged during the protocol execution. Our protocol ensures comprehensive protection at every stage: before, during, and after query processing.

The ideal function for a top-k free-connex conjunctive query (i.e., the CQ contains the relation schema, annotation information, output attributes $\boldsymbol{O}$) is presented in Algorithm 4, where we use $[\![\cdot]\!]$ to denote an element or a relation that is presented in secret-shared form.

Under the assumption that the servers do not collude, our protocol provides a strong guarantee: If the client is not corrupted, the adversary gains no knowledge beyond the public information (the input and output size, the relation schema). The intermediate join size (i.e., $|R_i \bowtie R_j|$ for some i, j), and even the full join size and project size (i.e., $|\mathcal{J}|$ and $|\mathcal{Q}|$) are protected. If the client is also corrupted, the only extra information the adversary learns about the honest parties is the final query result. From the adversary's perspective, all transcripts received during the execution of the protocol appear random, providing no discernible information about the underlying data.

Our secure building blocks have corresponding malicious versions, meaning our protocol can be extended to operate in the malicious setting. This extension would incorporate additional security measures and techniques to mitigate the risks posed by adversaries with malicious intent. We defer the detailed formal constructions and rigorous proofs to future work.

6 EXPERIMENTS

6.1 System Architecture

We have designed λSQL, an end-to-end framework based on our newly proposed algorithms. λSQL comprises two primary components: a **parser** and a **query rewriter**. The parser, adapted from [22], first interprets a given SQL query from the user and then generates a candidate join tree. This tree is subsequently fed into

the query rewriter. As detailed in Sections 3 and 4.2, the query rewriter transforms the join tree into revised SQL queries. These rewritten SQLs are automatically submitted to the chosen SQL engine for execution. Currently, λSQL is integrated with DuckDB, and the execution of rewritten SQLs on other platforms is conducted manually. However, it is designed to integrate smoothly with any existing SQL engine. By comparing these rewritten queries with the original SQL queries, we can effectively demonstrate the improved efficiency of our algorithms.

Additionally, we have developed a prototype system, SecTopK, focusing on secure query processing. This system is built upon the ABY3 framework[52] and incorporates the **level-k** and **product-k** algorithms, along with other fundamental operators like $\lambda_k^{\leq}$. While the query plans are currently drafted manually for each SQL query, our newly introduced APIs offer a declarative interface similar to platforms like Spark or Flink.

Optimization. An important note in Algorithm 1 is the repetitive processing of the relation S for $\log k$ iterations within the loop. To optimize performance and reduce redundant computations, our system opts to materialize this relation and construct an index on the join key before initiating the loop. This strategy has significantly enhanced the efficiency of **level-k**, accelerating computation by a factor of at least 5.

6.2 Experimental Setup

Query processing engines compared. To evaluate the efficacy of our optimized techniques, we selected PostgreSQL [2] and DuckDB[1] for centralized settings and Spark SQL [5] for parallel/distributed settings. All three engines are widely adopted in both academia and the industry. For secure computation, no established baseline exists for top-k queries. As an alternative, we use the time for sorting the full join results in ABY3 as our comparative baseline. Given that the baseline approach needs to sort the full join results to obtain the final top-k results, this runtime sets a lower bound for the baseline approach. Our experiments examined the single-thread and parallel efficiency of our algorithms on PostgreSQL, DuckDB, and Spark SQL. Meanwhile, we also compare our approach with state-of-the-art rank enumeration algorithm [23]. The source codes from [23] are written in C++ and cannot accept arbitrary SQL queries. To obtain a fair comparison, we also implemented our algorithms in C++ to remove the overhead caused by the database systems. To exclude I/O costs from the total execution time, we pre-loaded all data into memory and only measured the query execution time.

Experimental environment. Our experiments were executed on two machines. Experiments on PostgreSQL version 10.23, DuckDB version 0.6.1, and Spark SQL version 3.3.0 were conducted on a machine with dual Xeon 2.0GHz processors (28 cores/56 threads each), 1TB RAM, and running Oracle Linux 8.8. We used a machine with an Intel Core i5 3.0GHz processor, 6 cores/6 threads, 32GB RAM, and running MacOS 12 for secure query evaluations. We assigned 40 cores for Spark, while the other platforms used a single core during experiments. We executed each query 10 times on every engine, reporting the mean runtime. Each query runs at most 8 hours to obtain meaningful results.

Graph	#edge	#vertex	#Line-2	#Line-3	#Line-4	#Star	#Tree
Bitcoin	24186	3783	0.12	4.28	185	26.7	600
Wiki	103689	7115	0.45	20.2	914	479	19060
Epinions	508837	75879	3.99	372	37897	2826	168224
DBLP	1049866	13477	0.70	6.75	83.5	140	2365
Twitter	1768149	81306	13.3	1180	118615	3171	280886

Table 2: Graph datasets and their statistics. #edge is the input size of graph datasets. #Q is the full join size of Q over the corresponding graph datasets in units of 10 million ($\times 10^7$).

Datasets and Queries. We use graph pattern queries in the experiments with real-world graphs from SNAP (Stanford Network Analysis Project) [4], summarized in Table 2. We store edge information as a relation Graph(src, dst, rating) and the rating for each tuple is randomly generated. We evaluate 5 graph queries, including three line queries, Line-2, Line-3, and Line-4. For each query, we set $\otimes$ to be +, and $\oplus$ to be max. For example, the original SQL query for the Line-3 query is

```
SELECT R.src, R.dst, S.dst, T.dst,
  R.rating + S.rating + T.rating as total_rating
FROM graph R, graph S, graph T
WHERE R.dst = S.src and S.dst = T.src
ORDER BY total_rating DESC LIMIT k
```

We also include Star query and Tree query, where the original SQL for Tree query is

```
SELECT S.dst, S.src, R.src, T.dst, U.dst,
  R.rating+S.rating+T.rating+U.rating as total_rating
FROM graph R, graph S, graph T, graph U
WHERE R.dst = S.src and S.dst = T.src and S.dst = U.src
ORDER BY total_rating DESC LIMIT k
```

Due to the space constraint, the full list of original SQL queries, as well as optimized SQL queries after rewriting, are given in the code repository [3].

6.3 Experiment Results

Running time. Figure 5 presents the execution times across various engines. For these tests, we set $k = 1024$ for all queries, and $b = 32$ for **level-k** on plaintext. Bars touching the axis boundary indicate instances where the system either exceeded the 8-hour threshold or exhausted available memory.

Our optimization techniques significantly outperform original (vanilla) SQL. Across all tested queries and platforms, our rewritten SQLs consistently yield improvements ranging from 1 to 4 orders of magnitude. These techniques are particularly effective for queries generating large volumes of join results. For example, in the Tree query, we observed improvements of 2500x on SparkSQL, 20000x on DuckDB, and a remarkable 160000x on PostgreSQL.

Furthermore, our newly introduced algorithms have rendered secure query processing not only possible but also efficient. As illustrated in Figure 5, the improvements are more striking in scenarios requiring protected query procedures. For complex queries like Line-4, Star, and Tree, the ABY3 system fails to sort the full join results across all datasets. Interestingly, SecTopK not only succeeds in these cases but also surpasses the performance of vanilla DuckDB, which lacks security guarantees. However, it is important

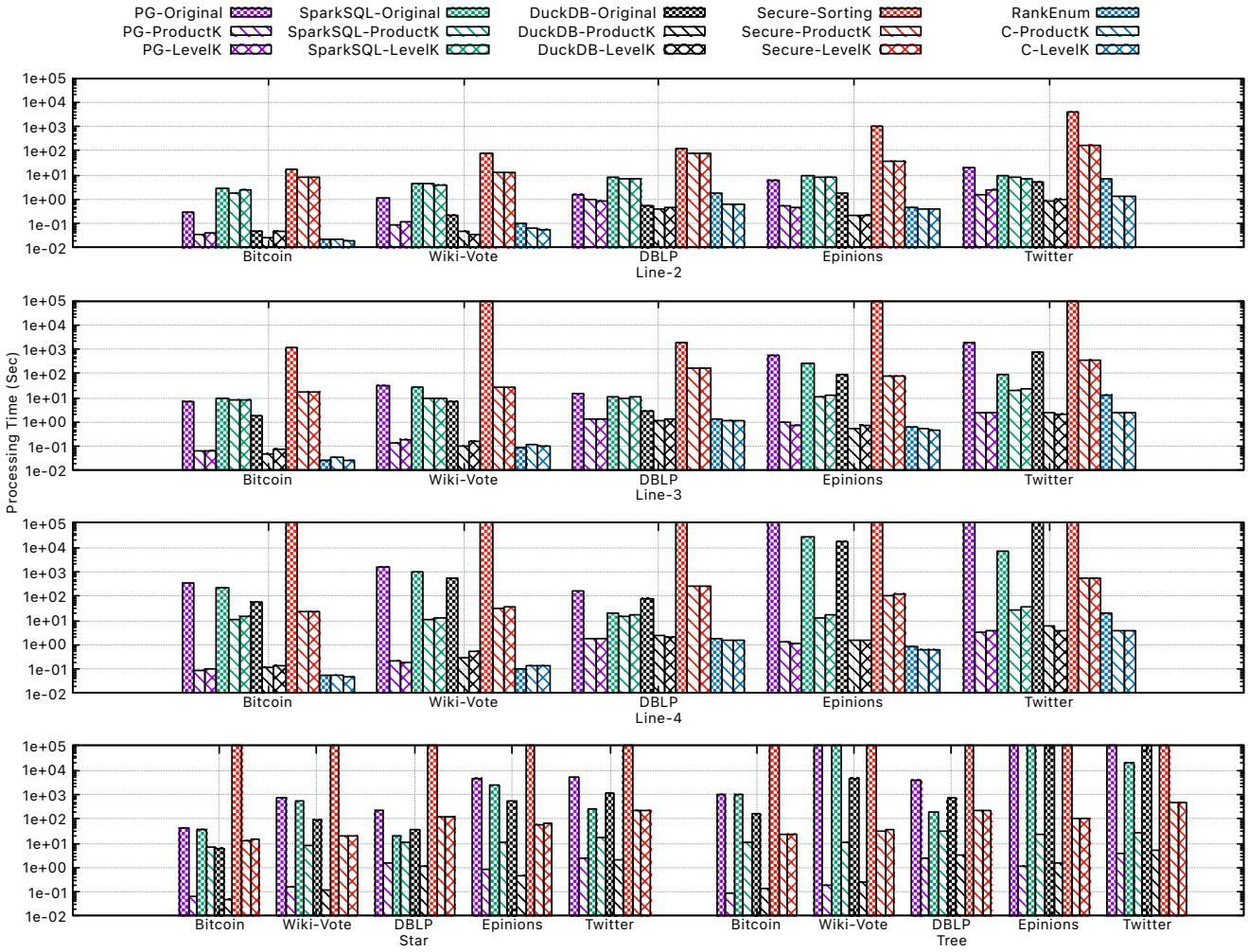


Figure 5: Running time of test queries.

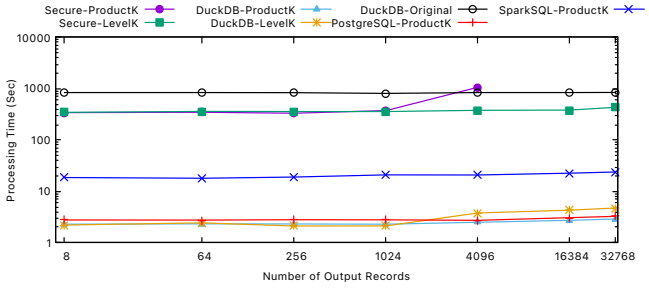


Figure 6: Running time v.s. different K (Line-3, Twitter).

to note that SecTopK requires 100x to 1000x more computation time compared to optimized plaintext SQL. This additional time is the cost of secure computation. Despite this, our innovative approach consistently completes all queries within a few hundred seconds, making the overall execution time both practical and acceptable.

Comparison with rank enumeration algorithm. We also compared both **level-k** and **product-k** with state-of-the-art rank enumeration algorithm from [23], with the results also illustrated in Figure 5. The performance difference is marginal for smaller datasets, but our algorithms demonstrates up to 7x improvement on larger datasets. This aligns with the analysis presented in Section 4.4, which indicates that our algorithms can improve performance by a $\log N$ factor compared to rank enumeration algorithms. The performance gain is less noticeable for smaller N but becomes more significant as N increases.

Breakdown time of λSQL. To further understand the efficiency of λSQL, we conducted experiments analyzing the time consumed by its various components, as shown in Table 3. Notably, the parsing and rewriting times are independent of the input size. However, there is potential to enhance λSQL's performance further. In typical database systems, optimization usually takes only a few milliseconds for a given query. Integrating λSQL directly into database systems could eliminate redundant parsing and enable the generation of execution plans without needing to rewrite queries, thus potentially boosting overall query execution efficiency. This integration would require customization for each database system, though. As a proof-of-concept prototype, λSQL is designed to be

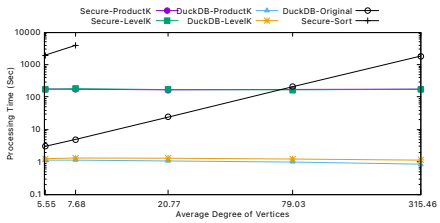


Figure 7: Running time v.s. degree (Line-3, DBLP).

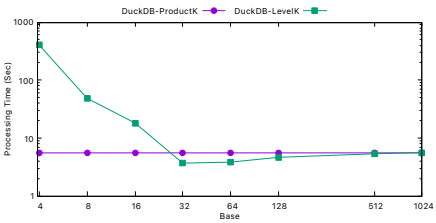


Figure 8: Running time v.s. base (Line-4, Twitter).

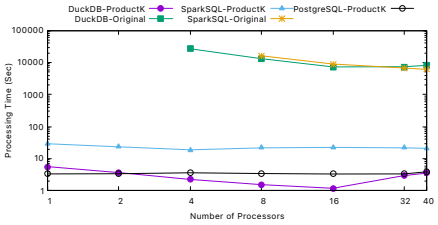


Figure 9: Running time v.s. parallelism (Line-4, Twitter).

Query	Parsing	Rewrite	I/O	Execution	Original
Line-3	0.058	0.002	0.612	2.379	1884.354
Line-4	0.064	0.003	0.612	3.325	\
Star	0.053	0.002	0.612	2.460	529.256
Tree	0.064	0.003	0.612	3.540	1128.236

Table 3: Breakdown of Running Time (s) of λSQL (Twitter).

easily integrated into any SQL-supporting database system, allowing for the testing and performance evaluation of **product-k** and **level-k** algorithms.

Impact of different k. As suggested by our theoretical results, the parameter k distinctly influences the performance of both **product-k** and **level-k** while its impact on the baseline approach is marginal. On the other hand, the experimental results from Figure 5 suggest both **product-k** and **level-k** have similar performance. We further studied this phenomenon by investigating the effects of varying k values for Line-3 query on the Twitter graph, with outcomes illustrated in Figure 6.

For the baseline approach, an increase in k doesn't alter its running time, given that the full join computation dominates its performance. On the other hand, as k increases, **level-k**'s runtime experiences a slight rise in both secure and plaintext computations. **product-k** exhibits a more significant increase in its execution time for secure computation, particularly when k exceeds 1024. Yet, for plaintext computation, its performance remains relatively stable. This discrepancy arises because secure computations require worst-case running times to ensure data independence. When $k = 1024$, the term k^2 becomes the dominant factor influencing the runtime. The worst case rarely happens in plaintext scenarios, enabling **product-k** to operate more efficiently. Additionally, a growth in k leads to a rise in computational rounds, imposing a constant overhead on **level-k** that impacts its efficiency. To summarize, for secure computations, **level-k** is more favorable due to its better worst-case runtime compared to **product-k**. In plaintext scenarios, however, **product-k** typically delivers better performance.

Impact of data distribution. An important property for oblivious algorithms is data-independent. To verify the data independence of both **level-k** and **product-k**, we manipulated the average degree of the graph. This was done by retaining all edges while clustering its vertices. The experiment results are illustrated in Figure 7.[7] For plaintext computations, the running times of both **level-k** and **product-k** decreased when the average degree increased. In contrast, the baseline approach sees linear growth in its running time as the full join results also increased when the average degree increased. When considering secure computations, however, the running time for both algorithms remains consistent due to the data-independence.

Impact of different base. When designing the **level-k** algorithm, we set the base $b = 2$. As the base increases, the theoretical overhead also increases. However, at the same time, a larger base efficiently reduces the length of the SQL, leading to fewer constant overheads introduced by SQL parsing and query optimization. To study the impact of different bases, we modified the base of the **level-k** algorithm, and the experimental results are shown in Figure 8. We found the total time first decreases and then increases, as the base increases, and $b = 1024$ corresponds to the result of **product-k**. We leave the way to choose the best b as an open problem.

Impact of Different Parallelism. To assess the impact of varying degrees of parallelism, we assigned different numbers of cores to each system and re-executed the Line-4 query on the Twitter graph. The results are shown in Figure 9. Increasing the parallelism in PostgreSQL did not substantially alter the runtime. Even with 40 cores, the original query failed to compute within the time limit. Meanwhile, both the original queries and our algorithms benefited from increasing parallelism on other platforms, with the original queries showing a more noticeable improvement. This is likely because our algorithms have already considerably reduced the overall computational load.

7 CONCLUSION

In this research, we introduced two relational algorithms, **level-k** and **product-k**, specifically designed to evaluate top-k conjunctive queries efficiently. These algorithms are not only optimal in their performance but also data-independent, a necessary condition for secure computation. Furthermore, we developed two systems: λSQL, which facilitates plaintext computation of top-k queries and can be integrated into any SQL-supporting database system, and SecTopK, dedicated to the secure computation of top-k queries. Our experimental results demonstrate substantial enhancements, with our new systems surpassing baseline performances by up to six orders of magnitude. We are optimistic that this work will stimulate further research in relational algorithms, paving the way for more straightforward implementation of novel algorithms in real-world systems and enabling swift adaptation of these algorithms for secure computing.

[7] Noted the running time for **level-k** and **product-k** are close, causing most of their data points to overlap.

REFERENCES

[1] DuckDB. https://duckdb.org/.

[2] PostgreSQL. https://www.postgre.org/.

[3] Relational Algorithms for Top-k Query Evaluation, Source Code Repository. https://anonymous.4open.science/r/TopKCQ.

[4] SNAP. https://snap.stanford.edu/snap/.

[5] SparkSQL. https://spark.apache.org/sql/.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Addison-Wesley Longman Publishing Co., Inc.

[7] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 13–28. https://doi.org/10.1145/2902251.2902280

[8] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105

[9] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 209–222.

[10] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.

[11] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proc. VLDB Endow.* 10, 6 (feb 2017), 673–684. https://doi.org/10.14778/3055330.3055334

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

[13] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. 1983. On the desirability of acyclic database schemes. *JACM* 30, 3 (1983), 479–513.

[14] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. 1–10.

[15] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Sławek Staworko, and Dominik Tomaszuk. 2022. Threshold Queries in Theory and in the Wild. *Proc. VLDB Endow.* 15, 5 (jan 2022), 1105–1118. https://doi.org/10.14778/3510397.3510407

[16] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.* 13, 1 (jan 2000), 143–202.

[17] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Trans. Database Syst.* 46, 2, Article 5 (may 2021), 41 pages. https://doi.org/10.1145/3450263

[18] Sunoh Choi, Gabriel Ghinita, Hyo-Sang Lim, and Elisa Bertino. 2014. Secure kNN Query Processing in Untrusted Cloud Environments. *IEEE Transactions on Knowledge and Data Engineering* 26, 11 (2014), 2818–2831. https://doi.org/10.1109/TKDE.2014.2302434

[19] Yannis Chronis, Thanh Do, Goetz Graefe, and Keith Peters. 2020. External Merge Sort for Top-K Queries: Eager Input Filtering Guided by Histograms. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2423–2437. https://doi.org/10.1145/3318464.3389729

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[22] Binyang Dai, Qichen Wang, and Ke Yi. 2023. SparkSQL+: Next-Generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 115–118. https://doi.org/10.1145/3555041.3589715

[23] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2022. Ranked Enumeration of Join Queries with Projections. *Proc. VLDB Endow.* 15, 5 (jan 2022), 1024–1037. https://doi.org/10.14778/3510397.3510401

[24] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *24th International Conference on Database Theory (ICDT 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[25] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. 2004. SQL:2003 Has Been Published. *SIGMOD Rec.* 33, 1 (mar 2004), 119–126. https://doi.org/10.1145/974121.974142

[26] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. *A Pragmatic Introduction to Secure Multi-Party Computation*.

[27] R. Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30, 3 (1983), 514–550.

[28] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Santa Barbara, California, USA) *(PODS '01)*. Association for Computing Machinery, New York, NY, USA, 102–113. https://doi.org/10.1145/375551.375567

[29] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and Efficient Algorithms for Rank Join Evaluation. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.org/10.1145/1559845.1559890

[30] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 218–229.

[31] Michel Gondran and Michel Minoux. 2008. *Graphs, dioids and semirings: new models and algorithms*. Vol. 41. Springer Science & Business Media.

[32] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. *arXiv preprint arXiv:2303.02723* (2023).

[33] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. 2021. Complexity Analysis of Generalized and Fractional Hypertree Decompositions. *J. ACM* 68, 5, Article 38 (sep 2021), 50 pages. https://doi.org/10.1145/3457374

[34] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 1999. Hypertree Decompositions and Tractable Queries. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (, Philadelphia, Pennsylvania, USA,) *(PODS '99)*. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/303976.303979

[35] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Information Security and Cryptology – ICISC 2012*. 202–216.

[36] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1740–1753.

[37] Zhian He, Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, Siu Ming Yiu, and Eric Lo. 2015. SDB: A Secure Query Processing System with Data Interoperability. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1876–1879.

[38] Xiao Hu and Qichen Wang. 2023. Computing the Difference of Conjunctive Queries Efficiently. *Proc. ACM Manag. Data* 1, 2, Article 153 (jun 2023), 26 pages. https://doi.org/10.1145/3589298

[39] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1259–1274. https://doi.org/10.1145/3035918.3064027

[40] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Supporting Top-K Join Queries in Relational Databases. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 754–765.

[41] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv.* 40, 4, Article 11 (oct 2008), 58 pages. https://doi.org/10.1145/1391729.1391730

[42] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[43] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Trans. Database Syst.* 45, 4, Article 17 (dec 2020), 41 pages. https://doi.org/10.1145/3426865

[44] Paraschos Koutris, Tova Milo, Sudeepa Roy, and Dan Suciu. 2015. Answering Conjunctive Queries with Inequalities. In *18th International Conference on Database Theory (ICDT 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[45] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2132–2145. https://doi.org/10.14778/3407790.3407814

[46] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. 2005. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/1066157.1066173

[47] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA,*

April 17-19, 2023, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 1031–1056. https://www.usenix.org/conference/nsdi23/presentation/liagouris

[48] Andrea Lincoln, Virginia Vassilevska Williams, and Ryan Williams. 2018. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1236–1252.

[49] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, Jonas Traub, and Volker Markl. 2021. What is the Price for Joining Securely? Benchmarking Equi-Joins in Trusted Execution Environments. *Proc. VLDB Endow.* 15, 3 (nov 2021), 659–672. https://doi.org/10.14778/3494124.3494146

[50] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W. Cheung. 2007. Efficient Top-k Aggregation of Ranked Inputs. *ACM Trans. Database Syst.* 32, 3 (aug 2007), 19–es. https://doi.org/10.1145/1272743.1272749

[51] Xianrui Meng, Haohan Zhu, and George Kollios. 2018. Top-k Query Processing on Encrypted Databases with Strong Security Guarantees. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 353–364. https://doi.org/10.1109/ICDE.2018.00040

[52] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 35–52.

[53] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1271–1287.

[54] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting Incremental Join Queries on Ranked Inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 281–290.

[55] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (mar 2018), 40 pages. https://doi.org/10.1145/3180143

[56] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *Proceedings of the 30th Conference on USENIX Security Symposium*.

[57] Amir Shpilka. 2003. Lower bounds for matrix product. *SIAM J. Comput.* 32, 5 (2003), 1185–1200.

[58] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *Proc. VLDB Endow.* 13, 9 (may 2020), 1582–1597. https://doi.org/10.14778/3397230.3397250

[59] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2659–2665. https://doi.org/10.1145/3318464.3383132

[60] Denny Vrandečić. 2012. Wikidata: A New Platform for Collaborative Data Collection. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) *(WWW '12 Companion)*. Association for Computing Machinery, New York, NY, USA, 1063–1064. https://doi.org/10.1145/2187980.2188242

[61] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *Proc. VLDB Endow.* 16, 5 (jan 2023), 1046–1058. https://doi.org/10.14778/3579075.3579080

[62] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3514221.3517830

[63] Qichen Wang and Ke Yi. 2023. Conjunctive Queries with Comparisons. *SIGMOD Rec.* 52, 1 (jun 2023), 54–62. https://doi.org/10.1145/3604437.3604450

[64] Yilei Wang and Ke Yi. 2022. Query Evaluation by Circuits. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/3517804.3524142

[65] Wai Kit Wong, David Wai-lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure KNN Computation on Encrypted Databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/1559845.1559862

[66] Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, and Siu Ming Yiu. 2014. Secure Query Processing with Data Interoperability in a Cloud Database Environment. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 1395–1406.

[67] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

[68] Andrew C Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 160–164.

[69] Bin Yao, Feifei Li, and Xiaokui Xiao. 2013. Secure nearest neighbor revisited. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 733–744. https://doi.org/10.1109/ICDE.2013.6544870

[70] Zhilin Zhang, Ke Wang, Chen Lin, and Weipeng Lin. 2018. Secure Top-k Inner Product Retrieval. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. Association for Computing Machinery, New York, NY, USA, 77–86. https://doi.org/10.1145/3269206.3271791

A MISSING PROOF IN SECTION 2

Proof of Theorem 2.1. Given two $n \times n$ Boolean matrices M_1, M_2, we can encode them with relations R and S, and assign all tuples with weight $v(t) = 1$. If such an algorithm exists, then we can set $k = n^2$ and evaluate the query. Since the input size $N \leq n^2$, the query can be evaluated with the algorithm in $\tilde{O}(n^2)$ time. The output corresponds to $M_1 \times M_2$, contravening the BMM conjecture. □

B MISSING PROOF IN SECTION 3

Proof of Theorem 3.1. Let $R_E = \pi^{\oplus}_E(R)$. For any $t \in R_E$, if $t \notin \lambda^{\lesssim}_k(R_E)$, then there exists $t_1, \ldots, t_k \in R_E$ such that $v(t) \prec v(t_i)$ for all $i \in [k]$. Since the annotations of R_E are obtained by computing the maximum annotations for each group of tuples in R, there exists $t'_1, \ldots, t'_k \in R$ such that $t'_i.E = t_i$ and $v(t'_i) = v(t_i)$. Therefore, for any $t' \in R$ with $t'.E = t$, $v(t') \preceq v(t) \prec v(t_i) = v(t'_i)$ for all $i \in [k]$. This means that any t' with $t'.E = t$ must not appear in $\lambda^{\lesssim}_k(R)$, hence $t \notin \pi^{\oplus}_E(\lambda^{\lesssim}_k(R))$, which concludes the theorem. □

C MISSING MATERIALS IN SECTION 4

Proof of Theorem 4.5. Firstly, we demonstrate correctness by asserting that for any $i \geq 0$, the condition $T_i = \lambda^{\lesssim}_k(\sigma_{ID \leq 2^{i+1}}(R \bowtie S))$ is true. This is substantiated via induction. It is obviously true for $i = 0$. Suppose the statement holds for $i \leq i_0$, and then for $i = i_0 + 1$,

$$\begin{aligned}
&\lambda^{\lesssim}_k(\sigma_{ID \leq 2^{i+1}}(R \bowtie S)) \\
=&\lambda^{\lesssim}_k\left(\lambda^{\lesssim}_k(\sigma_{ID \leq 2^{i}}(R \bowtie S)) \cup \lambda^{\lesssim}_k(\sigma_{2^i < ID \leq 2^{i+1}}(R \bowtie S))\right) \\
=&\lambda^{\lesssim}_k\left(T_{i-1} \cup \lambda^{\lesssim}_k(\sigma_{2^i < ID \leq 2^{i+1}}(R \bowtie S))\right).
\end{aligned}$$

If a tuple $(a, b, c, ID = k) \in T_i$ with $k > 2^i$ exists, there must also be a tuple $(a, b, c', ID = 2^i) \in T_i$ as $v(a, b, c') \succ v(a, b, c)$. Thus $(a, b) \in \pi_{A.B}(\sigma_{ID=2^i}(T_{i-1})) = R_i$, allowing for the extension of the previous equation to:

$$\lambda^{\lesssim}_k(\sigma_{ID \leq 2^{i+1}}(R \bowtie S)) = \lambda^{\lesssim}_k\left(T_{i-1} \cup \lambda^{\lesssim}_k(\sigma_{2^i < ID \leq 2^{i+1}}(R_i \bowtie S_i))\right) = T_i.$$

Subsequently, we explore the running time. Note that S_0 is computed by selecting tuples in S with ID 1 or 2, making the degree of S_0 on B bounded by two. Thus, each tuple in R can join a maximum of two tuples in S_0, ensuring $|R \bowtie S_0| \leq 2N$, thereby taking $O(N)$ time. Analogously, for any i, as the degree of S_i on B is 2^i, and once $|R_i| \leq k/2^i$ can be validated, then $|R_i \bowtie S_i| \leq k$, and making the running time of Line 5–7 $O(k)$.

To demonstrate $|R_i| \leq k/2^i$, consider any $(a, b, c, 2^i) \in T_{i-1}$. Since $T_{i-1} = \lambda^{\lesssim}_k(\sigma_{ID \leq 2^{i+1}}(R \bowtie S))$, a tuple $(a, b, c_j, j) \in T_{i-1}$ must exist for all $j \leq 2^i$, because $v(a, b, c_j, j) = v(a, b) \otimes v(b, c_j, j) \succ v(a, b) \otimes v(b, c, 2^i) = v(a, b, c, 2^i)$. Consequently, $|\sigma_{ID=2^i}(T_{i-1})| \leq k/2^i$, leading to $|R_i| \leq k/2^i$.

Given that Lines 5–7 execute a maximum of $\log k$ rounds, the total running time culminates to $O(N + k \log k)$. If base b exceeds 2, the maximal degree of S_i per round becomes b^i, with total rounds to be $\log_b k$, leading to a running time of $O(bN + bk \log_b k)$. □

Missing proof of Lemma 4.7. We can rewrite T as follow:

$$T := \lambda^{\lesssim}_k\left(\pi^{\oplus}_{E \cup E_p}(Q)\right) = \lambda^{\lesssim}_k\left(R \bowtie \pi^{\oplus}_{E_p}\left(\bowtie_{R' \in \boldsymbol{R} - \{R\}} R'\right)\right).$$

The aggregation $R'_p := \pi^{\oplus}_{E_p}(\bowtie_{R' \in \boldsymbol{R} - R} R')$ is a free-connex CQs. Notably, the query can be efficiently evaluated in $O(N)$ time, facilitated by the presence of E_p in a single relation R_p [10, 42]. Gottlob et al. [32] also shows how to rewrite such aggregation queries in SQL while preserving the running time. This is achieved by a series of group-by-aggregate and semi-join operators to represent such queries. After computing R'_p, an intriguing observation surfaces: the query becomes a top-k binary join query $T := \lambda^{\lesssim}_k(R \bowtie R'_p)$. Subsequently, we can use either **level-k** or **product-k** to solve the query in $O(N + k \log k)$ or $O(N + k^2)$ time. □

C.1 Unions and Outer Joins.

We now extend our algorithms to support unions and various types of join operators.

Unions. A Union of Conjunctive Queries (UCQ) is of the form $Q := Q_1 \cup \cdots \cup Q_m$, where each Q_i shares the same output attributes. An important property for top-k UCQ $\lambda^{\lesssim}_k(Q)$ is as follows:

Lemma C.1. $\lambda^{\lesssim}_k(Q) = \lambda^{\lesssim}_k(\bigcup_{i \in [m]} Q_i) = \lambda^{\lesssim}_k(\bigcup_{i \in [m]}(\lambda^{\lesssim}_k(Q_i)))$.

This lemma holds because for each Q_i, any query result not within the top-k of Q_i cannot be in the top-k of Q. Utilizing this lemma, we can evaluate any top-k UCQ with the following algorithm:

Algorithm 5: Evaluating top-k UCQs

Input: A UCQ $Q := \cup_{i \in [m]} Q_i$.
Output: Top-k result of $Q(D)$
// Compute $\lambda^{\lesssim}_k(Q_i)$ for each $i \in [m]$.
1 **forall** $i \in [m]$ **do**
2 $R_i \leftarrow \lambda^{\lesssim}_k(Q_i)$ // Using Algorithm 3 to compute.
// Union all query results and get the final top-k for Q.
3 $Q' := \lambda^{\lesssim}_k(\cup_{i \in [m]} R_i)$;
4 **return** Q'

Each individual Q_i is computed in $O(N^{w_i} + k \log k)$ time using **level-k** (or $O(N^{w_i} + k^2)$ using **product-k**). The for-loop in Algorithm 5 thus takes a total time of $O(N^w + k \log k)$, considering the query size m as constant and $w = \max_{i \in [m]} w_i$. Merging and sorting m tables with a total of km records can be done in $O(k \log k)$ time.

Theorem C.2. *Top-k UCQ Q can be solved in $O(N^w + k \log k)$ using **level-k** or $O(N^w + k^2)$ using **product-k**, where $w = \max_{i \in [m]} w_i$.*

Outer Joins. Our previous discussions have primarily focused on natural joins. We are now considering extending this to include outer joins. To support outer joins, we need first to define an appropriate annotation Annot for NULL values, as outer joins produce results even when there are no matching tuples in the other join table. For a left outer join $R \ltimes S$, we create a special tuple

NULL $\rightarrow$ Annot that can join with any tuples in R. We then rewrite the top-k queries as follows:

$$\lambda_k^{\preceq}(R \overrightarrow{\bowtie} S) = \lambda_k^{\preceq}(\lambda_k^{\preceq}(R \bowtie S) \cup \lambda_k^{\preceq}((R - R \ltimes S) \bowtie \{\text{NULL}\}).$$

Here, $R - R \ltimes S$ can be computed using the NOT EXISTS clause in SQL, and this operation can be executed in $O(N)$ time. Additionally, joining with a single tuple NULL can also be achieved in $O(N)$ time. Thus, a left outer join between R and S can be completed in $O(N + k \log k)$ time. The support for the right outer join is symmetrical to that of the left outer join. For full outer joins, we have:

$$\lambda_k^{\preceq}(R \overrightarrow{\bowtie\subset} S) = \lambda_k^{\preceq}(\lambda_k^{\preceq}(R \overrightarrow{\bowtie} S) \cup \lambda_k^{\preceq}(R \bowtie\subset S)).$$

Thereby, we can extend support to outer joins while maintaining the same complexity as natural joins.

Theta Joins. Theta joins represent another join where the join condition can be an arbitrary function between two tables R and S, in addition to equality conditions. Evaluating theta joins in $O(N^w + OUT)$ time remains an open problem, as does the efficient support for top-k theta joins. Tziavelis et al. [59] explored the support for rank enumeration in theta joins over comparison conditions like $R.A > S.B$. However, this approach does not extend to more complex comparison queries such as $R \bowtie S \bowtie T$ where $R.A > T.B$, and the generalization to support an arbitrary join function $f(R, S)$ is still an open question. We believe that will be an interesting direction to explore in the future research.