Secure Query Processing with Linear Online Cost

Qiyao Luo
OceanBase, Ant Group
luoqiyao.lqy@antgroup.com

Yilei Wang
Alibaba Cloud
fengmi.wyl@alibaba-inc.com

Wei Dong
NTU
wei_dong@ntu.edu.sg

Ke Yi
HKUST
yike@cse.ust.hk

***Abstract*—Query processing under the secure multi-party computation (MPC) model has received increasing attention in recent years. However, all existing MPC query processing algorithms incur a cost of $\Omega(n \log n)$, due to the use of secure sorting. While secure sorting is believed to be inevitable, we observe that it can be moved to a one-time preprocessing stage. By doing so, we manage to reduce the online cost of each subsequent query to linear for free-connex queries, a large class of select-project-join-aggregation queries. This matches the plaintext result for query processing, as free-connex queries are the largest class of queries known to be solvable in linear time on plaintext. We have then built SLOOP, a Secure Linear Online cOst query Processing system. Experimental results show that SLOOP significantly outperforms state-of-the-art methods, while supporting a much broader class of queries.**

***Index Terms*—secure multi-party computation, query processing, free-connex query**

I. Introduction

With significant improvements in both computing infrastructure and protocol designs, *secure multi-party computation (MPC)* is now transforming from a theoretical concept to a practical technique for breaking the barrier between information silos. It holds the appealing guarantee that a query can be evaluated on the private data of multiple parties without revealing anything beyond the query result. This meets the needs of enterprises who want to perform collaborative analysis without sharing their private data.

Example 1. Consider a scenario where an insurance company, a hospital, and a bank would like to collaboratively analyze customers' behaviour based on their insurance plans, medical history, and bank balances. Assume the following simplified schema: insurance plans `I(id, ratio)`, medical records `M(id, disease, cost)`, and bank balances `B(id, balance)`. The query below conducts a typical collaborative analysis, which finds the total insurance payment for patients whose balances are above a certain threshold, classified by diseases:

```
SELECT disease, SUM(M.cost * I.ratio)
  FROM I JOIN M ON I.id = M.id
         JOIN B ON M.id = B.id
 WHERE B.balance >= 10000
 GROUP BY disease
```

This is a typical *free-connex query* (formal definition in Section II-F) which can be executed in linear time on plaintext.

In real-world scenarios, these data are confidential due to privacy concerns and regulations requirements, which should not be revealed to others. A query processing system using MPC thus addresses this challenge. In this paper, we are interested in developing MPC protocols for such queries that can also achieve linear cost, including both running time and communication. △

The major technical challenge in query processing is the join operator. The brute-force method for computing a join of k relations, each having n tuples, is *nested-loop join*. It has a cost of $O(n^k)$, which is actually worst-case optimal as the output size of the join, where n denotes the input size and m the output size. However, worst-case optimality is not meaningful for joins, as $m \ll n^k$ on typical inputs. Thus, the database literature is more interested in an $O(n+m)$ running time. Such a running time is often called *linear*, which is clearly optimal. One landmark result in query processing is that linear time is achievable if and only if the query is *free-connex* [1]–[4]. The upper bound uses hash-joins and the Yannakakis algorithm, and the lower bound is based on some standard complexity assumptions [4].

Existing works on MPC joins have mostly focused on two-way joins, as summarized in Table I. We see that all protocols incur $\Omega(n \log n)$ cost due to the use of secure sorting, with the exception of FDJ [5], but it only supports the restricted case where the join key is unique in both input relations, also known as PK-PK joins. More critically, no practical protocols for general joins (i.e., no unique constraints on the join key) extend to multi-way joins. This severely limits our ability to ask complex analytical queries on private data.

A. Our Contributions

While secure sorting is believed to be unavoidable for general joins under MPC, we observe that it can be moved to a one-time preprocessing stage, after which the online cost per query can be reduced to linear. Specifically, we achieve the following results in this paper:

1) We propose a novel secure join protocol for general joins that achieves $O(n+m)$ online cost in both communication and computation, after an $O(n \log n)$ preprocessing.
2) We extend the protocol to support all free-connex queries, still with $O(n+m)$ online cost, which matches the plaintext result for the query processing problem. This is the first practical MPC protocol that supports free-connex queries on secret-shared input and output.
3) We build a system prototype SLOOP (Secure Linear Online cOst query Processing system) based on the new protocols. Experiments show that SLOOP outperforms

TABLE I: Comparison between SLOOP and previous works, where n and m are the input and output size of the query.

Method	Model	Two-way join	Free-connex queries on k relations
Senate [6] †	MPC	$O(n \log n)^{\star}$	Unsupported
FDJ [5]	3PC	$O(n)^{\star}$	Unsupported
SSA [7]	3PC	$O(n \log n)^{*}$	Unsupported
SSJ [8]	3PC	$O((n+m) \log n)$	Unsupported
Scape [9]	3PC	$O(n \log^2 n + m)$	Unsupported
OptScape	3PC	$O(n \log n + m)$	Unsupported
SMCQL [10]	2PC	$O(n^2)$	$O(n^k)$
Secrecy [11]	3PC	$O(n^2)$	$O(n^k)$
QCircuit [12]	MPC	$O((n+m) \log^4 n)$	$O((n+m) \log^4 n)$
SECYAN [13] ‡	2PC	$O(n \log n + m)$	$O(n \log n + m)$
SLOOP §	3PC	$O(n+m)$	$O(n+m)$

†Subject to plaintext input.
‡Subject to plaintext input and output.
§With $O(n \log n)$ preprocessing cost.
*Limited to PK-PK join, i.e., the join key on both input relations is unique.
*Limited to PK-FK join, i.e., the join key in at least one of the input relations is unique.

the state-of-the-art systems significantly, while supporting a much broader class of queries.

Our protocols work in the 3PC model with secret-shared input and output (detailed definition given in Section II-B). This model supports any number of data owners, allows easy deployment in the cloud, and can be composed with other 3PC protocols. It has gained significant popularity in recent years, and has been adopted by most recent works (see Table I).

B. Technical Overview

Plaintext join algorithms achieve linear cost by hashing, which is difficult to implement in MPC. This is because it assigns tuples to a hash table of multiple buckets, resulting in uneven, input-dependent bucket sizes. To guarantee security, all buckets should be padded to a high-probability size upper bound, which severely downgrades the efficiency. As a result, all existing efficient MPC join protocols [6], [8], [9] are based on sort-merge-join, which inevitably incurs an $\Omega(n \log n)$ cost. Our key observation is that the sorting can be moved to a one-time preprocessing stage, which yields the *ranks*, an extra attribute storing the position after sorting (detailed definition in Section III-A), of each input relation. During online time, these ranks allow us to replace sorting with permutation, which has linear cost and runs in a constant number of rounds under 3PC. We note that using preprocessing to speed up online processing is actually a common and useful technique in both secure computation (e.g., private information retrieval [14]–[17], Beaver triples [18], and secure sampling [19]) and plaintext query processing (i.e., indexing), but this is the first time it is applied to secure query processing.

When going from two-way joins to multi-way joins, we face another difficulty. The preprocessing can only provide the ranks for the base relations, but not the intermediate join results. For example, consider a simple 3-way join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$ with the query plan: (1) $R_{12} \leftarrow R_1 \bowtie R_2$; (2) $T \leftarrow R_{12} \bowtie R_3$. The precomputed ranks of $R_1.B$ and $R_2.B$ allow us to compute the first join in linear cost, but we do not have the ranks of $R_{12}.C$ to speed up the second join. This is because R_{12} is ordered by B after the first join, and the original rank of $R_2.C$ is invalid since the tuples in R_{12} are duplicated or eliminated because of the join. To address this issue, the first join should not just compute R_{12}, but also its ranks, namely, we need to make the ranks *transferable*. More importantly, the transformation must also be done with linear cost in order to keep the total cost linear. In fact, the bulk of our technical development is on how to transfer the ranks through each of the standard relational operators: join $\bowtie$, semi-join $\ltimes$, selection σ, and projection/aggregation Π. This allows us to handle any query plan $\mathcal{P}$ that consists of these operators. Due to the security requirement, the size of each intermediate relation in $\mathcal{P}$ must be padded to its worst-case size. The well-known Yannakakis algorithm [1] generates a query plan for any free-connex query with worst-case size equal to $O(n+m)$, thus ensuring that the total cost of our protocol is also linear (Section IV).

In addition to guaranteeing the linear asymptotic bound, our system prototype SLOOP also takes the concrete costs into consideration for further optimization (Section VI-B), including faster rank generation on preprocessing step and query plan selection. For query plan selection, we adapt the standard query optimization pipeline to MPC. Specifically, for a free-connex query, we first enumerate all the query plans. Then we estimate the MPC cost of each plan using a cost model that we develop for MPC query processing, and choose the best one for execution. Finally, we plug our 3PC protocols for various relational operators into the chosen plan, and pass them to the execution engine.

C. Related Work

Table I summarizes existing MPC join protocols. The first three are specialized for either PK-PK join or PK-FK join. Senate [6] requires that each input relation is held by a party in plaintext, so that the party can sort its relation on the join key locally in plaintext. Then they use the bitonic merger circuit to merge the relations into a fully sorted list for further operations. FDJ [5] provides the first linear cost

PK-PK join (equivalently, private set intersection (PSI) with payload) protocol that accepts secret-shared input in 3PC. It also shows how to extend their protocol to support PK-FK join, but their extension reveals the degree information (i.e., the frequencies of the join attributes on the foreign-key side), thus not satisfying the security definition. A recent work SSA [7] presents a 3PC PK-FK join protocol with $O(n \log n)$ cost.

The two concurrent works of SSJ [8] and Scape [9] support a general join. They have achieved similar, but incomparable, results. However, we observe that Scape used the $O(n \log^2 n)$ bitonic sorter circuit instead of the optimal $O(n \log n)$ sorting algorithms [20], [21]. We have re-implemented Scape using the $O(n \log n)$ sorting protocol, and denote this optimized version as OptScape. OptScape is asymptotically better than both Scape and SSJ. It also matches SSA on PK-FK joins since $m = n$ on PK-FK joins. However, these protocols are only designed for a two-way join, and do not generalize to multi-way joins. Note that the direct composition of their two-way join protocols would reveal the intermediate join sizes instead of just the final output size, breaching the security guarantee. Recall the example of a simple 3-way join $(R_1 \bowtie R_2) \bowtie R_3$. By the security definition, only $m = |R_1 \bowtie R_2 \bowtie R_3|$ could be revealed, but their protocols would also reveal $|R_1 \bowtie R_2|$.

The next three protocols have designed circuits for joins, which can be directly translated into MPC protocols using generic techniques [22]. SMCQL [10] and Secrecy [11] are based on the nested-loop join, and their two-way join result size is padded to the worst case n^2, thus can be composed to support free-connex queries. Nevertheless, the cost becomes $O(n^k)$ which is impractical. QCircuit [12] proposes a circuit for join with size $O((n+m)\log^4 n)$ and has generalized the circuit to support free-connex queries with the same cost. However, the high polylogarithmic factor makes it of only theoretical interest.

The only practical protocol that supports free-connex queries is SECYAN [13], but it requires both the input and output to be in plaintext (more precisely, each party holds one input relation in plaintext, and the protocol outputs the query results to one party in plaintext), which greatly limits its applicability (see the discussion in Section II-B).

II. Preliminaries

A. Secret-Sharing

Secret sharing is an essential ingredient in many MPC protocols. We assume that each attribute of a tuple is represented by an ℓ-bit word. An (a, b)-*secret sharing scheme* splits an ℓ-bit secret v into b shares, such that any $a-1$ of the shares reveal no information about v, while any a shares allow the reconstruction of v.

In this paper, we adopt the replicated secret sharing [23], a (2,3)-secret sharing scheme, efficient for 3PC: For a given ℓ-bit secret value v, pick two random ℓ-bit numbers v_0, v_1 independently, and set $v_2 = v \oplus v_0 \oplus v_1$, where $\oplus$ denotes bit-wise XOR. Then the three shares are (v_0, v_1), (v_1, v_2), and (v_2, v_0), respectively. It is clear that any two shares reconstruct v, while any single share just consists of two information-less random numbers. We use $[\![v]\!]$ to denote v in secret-shared form and $[\![v]\!]_c$ to denote the share held by party c.

B. The 3PC Model

We adopt the standard three-party model with honest majority [5], [7]–[9], [11], [21], [23], [24], referred to as 3PC, which has gained significant popularity in recent years. There are three computing parties (also called "servers"), and at most one of them is corrupted. The query processing pipeline in the 3PC model works as follows: First, an arbitrary number of data owners upload their data to the three parties using a (2, 3)-sharing scheme. Then the parties perform some preprocessing on the secret-shared input. During online time, clients submit queries to the three parties, and they collaboratively compute the queries and return the results to the clients still in a (2, 3)-shared form.

In addition to being easy to deploy in the cloud (one can choose the three servers from three different cloud providers), 3PC protocols with secret-shared input allow the input relations to be partitioned arbitrarily – horizontally or vertically – to any number of data owners. Furthermore, outputting the query results in secret-shared form allows some further downstream processing. For example, one may compute a free-connex query and feed the query result to a 3PC machine learning protocol. Note that some existing protocols, such as Senate [6] and SECYAN [13], do not meet these requirements.

1) Security Definition: The security of a 3PC protocol is established by the real-ideal paradigm [22]: In the real world, the servers follow the prescribed protocol π over their secret-shared input $[\![x]\!]$ and get the secret-shared output $[\![y]\!]$. In the ideal world, there is a completely trusted party that collects all the inputs $[\![x]\!]$ from the servers to recover the input x, computes the output $y = \mathcal{F}(x)$ over the input, and delivers the secret-shared outputs $[\![y]\!]$ to the servers. The view of a party consists of its own input, output, and the transcript (for real world only), i.e., other messages it sent and received during the protocol. We assume a *semi-honest* adversary which can corrupt at most one party (i.e., the corrupted party will follow the protocol, but try to learn as much information as possible from its view). We say π realizes $\mathcal{F}$ if for the same input x, π and $\mathcal{F}$ will get the same output y. We say that π is secure against a semi-honest adversary if there exists a simulator that can simulate any party's real-world view of π by its ideal-world view of $\mathcal{F}$. The simulated real-world view should be indistinguishable from the party's true real-world view. Formally,

Definition II.1. Let $\mathcal{F}$ be a functionality, π be a protocol, x be an input of $\mathcal{F}$. Let $\mathrm{Real}_c(x) = ([\![x]\!]_c, [\![y']\!]_c, t_c(x))$ be the real-world view of a party c on input x with protocol π, where $[\![x]\!]_c, [\![y']\!]_c, t_c(x)$ are the party's shared input, shared output, and the transcript, respectively. Similarly, let $\mathrm{Ideal}_c(x) = ([\![x]\!]_c, [\![y]\!]_c)$ be the ideal-world view of a party c on input x with $y = \mathcal{F}(x)$. We say that π securely realizes $\mathcal{F}$ against a semi-honest adversary if $y' = y$ for all x, and there exists a simulation function $\mathrm{Sim}(\cdot)$ such that, for any x and

$c \in \{0,1,2\}$, the distributions of $t_c(x)$ and $\mathrm{Sim}_{\mathcal{F}}(\mathrm{Ideal}_c(x))$ are computationally indistinguishable in terms of κ, i.e., no polynomial-time (in κ) algorithm can distinguish them by probability more than $1/2 + \nu(\kappa)$, where κ is the *security parameter* [22] and $\nu(\kappa)$ is a negligible function of κ.

This definition thus captures the intuition that the real-world's transcript contains no information beyond the ideal-world view, which only contains information about the input and output sizes as both the input and output are given in secret-shared form.

2) Secure Query Processing: In the context of query processing, the definition of the functionality $\mathcal{F}$ (see Algorithm 1) consists of the database schema $\mathbf{R}$ (including the relation names and attribute names/types) and the query $\mathcal{Q}$. It reveals the exact output size m[1] (we show the secure computation of m in Section IV-C) and outputs the secret-shared query results.

Algorithm 1: Functionality $\mathcal{F}$ of query processing

```
Input: A query $\mathcal{Q}$; input relations $[\![R_1]\!], \cdots, [\![R_k]\!]$
Output: Query size $m$; query result $[\![T]\!]$
1 Recover $R_i$ from $[\![R_i]\!]$ for $1 \le i \le k$;
2 $T \leftarrow \mathcal{Q}(R_1, \dots, R_k)$;
3 Reveal $m \leftarrow |T|$;
4 Compute the secret share $[\![T]\!]$ of $T$;
5 return $[\![T]\!]$
```

C. Complexity Measures

Following the convention of prior works [5], [9], [11], [13], we measure the asymptotic complexity on the word level. This means that any standard arithmetic or logical operation between two ℓ-bit words is considered to have $O(1)$ cost in running time and communication (if needed). When the running time and communication are asymptotically the same, we use the term "cost" to refer to both.

For asymptotic complexity of query processing, we will assume that the query size (i.e., the number of relations and the number of variables/attributes in each relation) is a constant. In practice, these will also affect the concrete cost of a query plan, and is more accurately considered in Section VI.

Finally, in addition to running time and communication, the number of communication rounds is also important. All the protocols in this paper run in $O(\log n)$ rounds so the total latency overhead is negligible.

D. Dummy Tuples

For query processing on secret-shared data, we need to pad dummy tuples to intermediate results to protect relation sizes without letting the servers know whether a tuple is dummy or not. This is done by adding a *dummy marker* attribute to each intermediate relation. The value of a dummy marker is either 0 (dummy) or 1 (not dummy), which is also secret-shared to the three servers. In the description of the algorithms, we use $\perp$ to denote a dummy tuple. All dummy tuples are defined to be equal, while any non-dummy tuple is not equal to any dummy tuple.

[1]To protect m, the output size should be worst-case size n^k and a secure nested-loop join [10] is enough.

E. 3PC Primitives

We will make use of the following 3PC primitives, where the input and output of each primitive are both secret-shared. All primitives have been studied before, but we need to make some changes to the intersection and expansion primitive for our purpose.

1) Circuit-based computation: There are 3PC protocols [23] that, given two values x and y, output $x \oplus y$ (x, y, and $x \oplus y$ are all in secret-shared form; henceforth all data is secret-shared unless specified otherwise), where $\oplus$ can be any standard arithmetic (addition, multiplication, comparison, etc.) or logical operation (AND, OR, XOR). These protocols thus allow any circuit-based computation to be carried out in the 3PC model, with cost proportional to the size of the circuit and the number of rounds proportional to the depth.

In particular, we will make use of the *prefix sum* circuit: Given an associative operator $\oplus$ and an array $X = (x_1, \dots, x_n)$, this circuit computes $(x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \cdots \oplus x_n)$ in $O(n)$ size and $O(\log n)$ depth [25].

We will also need a slightly more general version, called *segmented prefix sum*. It takes two arrays $A = (a_1, \dots, a_n)$ and $X = (x_1, \dots, x_n)$ as input, where equal elements in A that appear consecutively define a segment. The outputs are the prefix sums of each segment in X. For example, if $A = (2, 2, 4, \perp, \perp)$, then the output is $(x_1, x_1 \oplus x_2, x_3, x_4, x_4 \oplus x_5)$. [12] shows how to adapt a prefix-sum circuit to compute segmented prefix sum still in $O(n)$ size and $O(\log n)$ depth.

There are two common choices for $\oplus$: When $\oplus$ is the arithmetic addition $+$ (define $x + \perp = \perp$), we simply call the output of (segmented) prefix sum as the `prefix_sum` of X (segmented by A). The other common choice is

$$x \oplus y := \begin{cases} x, & \text{if } y = \perp; \\ y, & \text{otherwise.} \end{cases}$$

This instantiation, which we call `prefix_copy`, replaces each dummy value in X with the last non-dummy value before it (if there is one).

2) Sorting: Sorting takes an array $X = (x_1, x_2, \cdots, x_n)$ and outputs an array $Y = (y_1, y_2, \cdots, y_n)$ in ascending order which is a permutation of X. In 3PC model, the state-of-art sorting protocols have cost $O(n \log n)$ [20], [21].

3) Permutation: Permutation takes two arrays $X = (x_1, \cdots, x_n)$ and $P = (p_1, \cdots, p_n)$ where P is a permutation of $[n] = (1, \cdots, n)$, and outputs an array $Y = (y_1, \cdots, y_n)$, where $y_{p_i} = x_i$ for all $i \in [n]$, i.e., each x_i is moved to position p_i in the output. We call the output Y the `permutation` of X specified by P. There is a 3PC permutation protocol with $O(n)$ cost and $O(1)$ rounds [26].

4) Compaction: This primitive takes two arrays $X = (x_1, \dots, x_n)$ and $T = (t_1, \dots, t_n)$ as input, where each $t_i \in \{0, 1\}$. Each x_i is said to be *marked* if $t_i = 1$. The `compaction` of X on T permutes X such that all marked elements appear before non-marked elements, and the relative ordering of the marked elements must be preserved. Compaction can be done by `prefix_sum` and `permutation` [21], hence it also has $O(n)$ cost and $O(\log n)$ rounds. We will use compaction to move all non-dummy tuples to the front of dummy tuples, where we use the dummy markers as T.

5) Intersection: Intersection takes two arrays $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_n)$ as input, where all non-dummy elements are distinct in X and Y, respectively. It outputs an array $T = (t_1, \dots, t_n)$ such that $t_i = 1$ if $x_i \neq \perp$ and $x_i = y_j$ for some j, otherwise $t_i = 0$. Furthermore, each y_j may have a payload v_j. In this case, the output should be $t_i = v_j$ if $x_i \neq \perp$ and $x_i = y_j$ for some j (note that there is at most one j such that $x_i = y_j$), otherwise $t_i = 0$. [5] provides a 3PC `intersection` protocol with payload that has $O(n)$ cost and $O(1)$ rounds.

We generalize their intersection protocol to allow duplicate elements in X, but the duplicates must be consecutive. We still require the elements in Y to be distinct so that the output is well-defined. This is how we handle duplicates in X: We first assign consecutive indices to each segment of equal elements in X. This can be done by computing the `prefix_sum` of $(1, 1, \dots, 1)$ segmented by X. Let r_i be the index of x_i. Next, we compute the `intersection` of $((x_1, r_1), \dots, (x_n, r_n))$ and $((y_1, 1), \dots, (y_n, 1))$, possibly with payload. For each segment in X, only the first element (i.e., the one with $r_i = 1$) gets the correct payload, while other elements get 0. Finally, we use segmented `prefix_sum` to copy the payload of the first element in each segment to the other elements. The protocol has $O(n)$ cost and $O(\log n)$ rounds, too. The generalized version is a sequential composition of prefix sum circuits and original `intersection` protocol. Therefore, its security is directly implied via composition theorem [27].

6) Expansion: Expansion takes two arrays $X = (x_1, \dots, x_n)$ and $D = (d_1, \dots, d_n)$ as input, together with a parameter m. It is guaranteed that (1) $x_i = \perp$ implies $d_i = 0$; and (2) $D_\Sigma := \sum_{i=1}^{n} d_i \leq m$. Each d_i, which we call the *degree* of x_i, is a non-negative integer that indicates the number of repetitions that x_i should appear in the output. The output is a length-m array:

$$(\underbrace{x_1, \dots, x_1}_{d_1 \text{ times}}, \underbrace{x_2, \dots, x_2}_{d_2 \text{ times}}, \dots, \underbrace{x_n, \dots, x_n}_{d_n \text{ times}}, \underbrace{\perp, \dots, \perp}_{m - D_\Sigma \text{ times}}).$$

We call the output array the `expansion` of X with degree D and output size m. [9] shows how this can be done with $O(n+m)$ cost and $O(\log(n+m))$ rounds under the constraints that $m = D_\Sigma$ and $d_i \neq 0$ for all i (hence X contains no dummies). Their algorithm works as follows:

1) Compute the first index where each x_i will appear in the output array. This can be done by computing the `prefix_sum` of $\{1, d_1, d_2, \dots, d_{n-1}\}$. Denote the result as $\{p_1, \dots, p_n\}$.
2) Distribute each x_i to index p_i. Let $Y = \{y_1, \dots, y_m\}$ be the resulting array. Any y_j such that $j \neq p_i$ for all i is dummy.
3) Run `prefix_copy` on Y.

Below we show how to remove the two constraints above. To remove the constraint $m = D_\Sigma$, we just need to set those elements in Y with positions larger than D_Σ to dummy after the final step, which can be done by a circuit of size $O(m)$ and depth $O(1)$. Removing the $d_i \neq 0$ constraint is trickier, since $d_i = 0$ means that x_i and x_{i+1} have the same target index, and step (2) above will fail. Our solution is that, after (1), we set p_i to $m + i$ if $d_i = 0$. Note that this guarantees that all p_i's are distinct. Then we run step (2) to output a length-$(m+n)$ array, where those x_i with $d_i = 0$ will be moved to index $p_i > m$. After step (2), we discard the last n elements, and run step (3). It should be clear that this modified algorithm still has $O(n+m)$ cost and $O(\log(n+m))$ rounds. Since the modifications are implemented in circuit-based computation using a length-$(m+n)$ array, the security still holds.

F. Free-connex Queries

For a select-project-join query $\mathcal{Q}$ in the form:

$$\mathcal{Q} = \Pi_{\boldsymbol{O}}\big(\sigma_{\gamma_1}(R_1(\boldsymbol{F}_1)) \bowtie \cdots \bowtie \sigma_{\gamma_k}(R_k(\boldsymbol{F}_k))\big),$$

its *free-connex join tree* $\mathcal{T}$ of $\mathcal{Q}$ is a tree that

1) the set of nodes are the relations $R_1, \dots, R_k$,
2) for any attribute $A \in \boldsymbol{F} = \boldsymbol{F}_1 \cup \cdots \cup \boldsymbol{F}_k$, the set of nodes (relations) that contain A are connected, and
3) for any $A \in \boldsymbol{O}$ and $B \in \boldsymbol{F} - \boldsymbol{O}$, $\mathsf{TOP}(B)$ is not an ancestor of $\mathsf{TOP}(A)$ in $\mathcal{T}$, where $\mathsf{TOP}(X)$ denotes the highest node in $\mathcal{T}$ containing attribute X.

The query $\mathcal{Q}$ is said to be *free-connex* if it has a free-connex join tree. Given a free-connex join tree, the classical Yannakakis algorithm [1] generates a query plan for any free-connex query in which each intermediate join has size bounded by the final query output size m. In fact, for general joins, free-connex queries are the largest class of queries that can be evaluated in $O(n+m)$ time on plaintext.

In the original definition of a free-connex query, the projection operator $\Pi_{\boldsymbol{O}}$ can also be replaced by some aggregation operator $\Pi^{\oplus}_{\boldsymbol{O}}$, called free-connex query with aggergation.

III. Ranks and Relational Operators

In this section, we first introduce ranks and how they are precomputed. Then we describe how to compute each operator in linear cost under 3PC with the help of ranks of the input relations. We will also show how the ranks are transferred from the input relations to the output relation. All input and output relations are secret-shared.

A. Ranks

Let $R(\boldsymbol{F})$ be a relation of size n with attributes $\boldsymbol{F}$. For any $\boldsymbol{E} \subseteq \boldsymbol{F}$, the rank of a tuple $t \in R$ is simply its position after sorting R by $\boldsymbol{E}$, ties broken arbitrarily and dummy tuples greater than non-dummy tuples. For each $\boldsymbol{E}$, these ranks form the *rank attribute* of $\boldsymbol{E}$, denoted as $\hat{\boldsymbol{E}}$. We use the notation $R(\boldsymbol{F}; \hat{\mathcal{E}})$ to denote a relation R augmented with the ranks $\hat{\mathcal{E}}$, which is secret-shared with the three servers, where $\hat{\mathcal{E}} = \{\hat{\boldsymbol{E}} \mid \boldsymbol{E} \subseteq \boldsymbol{F}, \boldsymbol{E} \in \mathcal{E}\}$ and $\mathcal{E} \subseteq 2^{\boldsymbol{F}}$ is a set of subsets of $\boldsymbol{F}$. The

TABLE II: Comparison among indexes, mapping, and ranks

	Indexes	Mapping [26]	Ranks
Setting	Plaintext	Secure	Secure
Reusable	✓	✓	✓
Transferable	×	×	✓
Operators	$\sigma, \ltimes, \bowtie$	Π	$\Pi, \ltimes, \bowtie$
Storage	$O(n)$	$O(n)$	$O(n)$
Preprocessing cost	$O(n)$	$O(n \log n)$	$O(n \log n)$

top row of Figure 1 shows an example where we augment a relation $R(A, B)$ with two rank attributes $\hat{A}, \hat{B}$.

A rank attribute $\hat{\boldsymbol{E}}$ can be generated by using an MPC sorting protocol to sort the input relation by $\boldsymbol{E}$ with $O(n \log n)$ cost. During preprocessing, we could generate the ranks for all the subsets of attributes, which is still a constant in terms of data complexity. Theoretically, there are $2^{|\boldsymbol{F}|} - 1$ non-empty subsets of a column set $\boldsymbol{F}$. However, our design exploits the observation that a rank generated for a composite key $\boldsymbol{E_1}$ can also support any query involving a prefix $\boldsymbol{E}_2$ of $\boldsymbol{E}_1$. For example, a rank of (A, B) can be effectively reused for queries on A, since tuples ordered by (A, B) are inherently sorted by A. This prefix-based reuse enables a significant reduction in the number of ranks needed. For instance, in a relation $R(A, B, C, D)$, only 6 carefully chosen ranks—such as (A, B, C, D), (A, C, D), (A, D, B), (B, C, D), (D, B) and (C, D)—are sufficient to support all 15 non-empty subsets through prefix coverage. In practice, we generate ranks only for columns that are semantically and syntactically likely to be used in join conditions or group-by clauses. For example, for TPC-H's `lineitem` relation, ranks are generated for `orderkey` and (`partkey`, `suppkey`) due to their role as foreign keys. No rank is generated for columns like `quantity` and `comment`, as they almost never appear in join conditions or group-by clauses. These ranks are stored in a rank cache. When a query arrives that depends on a ranking order not materialized during the offline phase, the system dynamically computes the required rank on-the-fly. Any rank computed during the online phase is also retained in the rank cache for future use. During query processing, only the columns and ranks explicitly required by the query are accessed and computed upon, which ensures that the computational overhead remains tightly coupled to the query's actual needs, rather than being influenced by the total number of ranks stored in the system.

1) Comparison among indexes, mapping, and ranks: We compare our proposed ranks with widely used indexes in plaintext database, as well as mapping [26] for secure graph analysis in Table II, as they are all auxiliary data that can be generated in preprocessing phase for query acceleration. As mentioned in Section I-B, ranks are transferable, but neither indexes or mapping is.

2) Relational operator with ranks: In the following sections we propose protocols for secure relational operators that (1) achieve linear costs using ranks and (2) generate ranks for the output relation with also linear costs. We summarize the relational operators that we can support in Table III.

TABLE III: Relational operators with ranks

Operator	σ	$\Pi_{\boldsymbol{E}}$	$\ltimes$	$\bowtie$
Input 1	$R(\boldsymbol{F}; \hat{\boldsymbol{E}})$	$R(\boldsymbol{F}; \hat{\mathcal{E}})$	$R(\boldsymbol{F_R}; \hat{\mathcal{E}}_R)$	$R(\boldsymbol{F_R}; \hat{\mathcal{E}}_R)$
Input 2	/	/	$S(\boldsymbol{F_S}; \hat{\mathcal{E}}_S)$	$S(\boldsymbol{F_S}; \hat{\mathcal{E}}_S)$
Output	$T(\boldsymbol{F}; \hat{\boldsymbol{E}})$	$T(\boldsymbol{E}; \hat{\mathcal{E}}')$	$T(\boldsymbol{F_R}; \hat{\mathcal{E}}_R)$	$T(\boldsymbol{F_R} \cup \boldsymbol{F_S}; \hat{\mathcal{E}}_R \cup \hat{\mathcal{E}}_S)$
Cost	$O(\lvert R \rvert)$	$O(\lvert R \rvert)$	$O(\lvert R \rvert + \lvert S \rvert)$	$O(\lvert R \rvert + \lvert S \rvert)$
Remark	/	$\boldsymbol{E} \in \mathcal{E}$ $\mathcal{E}' = \mathcal{E}_{\subseteq \boldsymbol{E}}$	$\boldsymbol{F_R} \cap \boldsymbol{F_S} \in \mathcal{E}_R$ $\boldsymbol{F_R} \cap \boldsymbol{F_S} \in \mathcal{E}_S$	$\boldsymbol{F_R} \cap \boldsymbol{F_S} \in \mathcal{E}_R$ $\boldsymbol{F_R} \cap \boldsymbol{F_S} \in \mathcal{E}_S$

B. Selection σ

The selection operator $\sigma_\gamma(R)$ returns the subset of tuples of R that pass the predicate γ, i.e., $\sigma_\gamma(R) := \{t \in R \mid \gamma(t) = \texttt{true}\}$. We assume that γ can be evaluated by a circuit of $O(1)$ size, so it takes $O(n)$ cost to evaluate γ on all tuples of R. However, since the results of the predicates are also secret-shared, the servers cannot and should not remove them physically. Instead, we set a tuple to dummy if it does not pass the predicate.

Since some tuples are turned into dummies after selection, existing ranks are no longer valid. Thankfully, since $\sigma_\gamma(R)$ is a subset of R, we can update the ranks by first permuting the tuples by the old ranks and then compacting out the dummy tuples in $O(n)$ cost , as shown in Algorithm 2.[2]

Algorithm 2: Update ranks

```
Input: Relation $R(\boldsymbol{F}; \hat{\mathcal{E}})$
Output: Relation $R(\boldsymbol{F}; \hat{\mathcal{E}})$ with updated ranks
1 for $\hat{\boldsymbol{E}} \in \hat{\mathcal{E}}$ do
2     $R \leftarrow$ permutation of $R$ by $R.\hat{\boldsymbol{E}}$;
3     $R \leftarrow$ compaction of $R$;
4     for $i \leftarrow 1$ to $n$ do in parallel
5         $R[i].\hat{\boldsymbol{E}} \leftarrow i$;
6 return $R$
```

C. Projection/Aggregation Π

1) Projection: Let $\Pi_{\boldsymbol{E}}$ be the projection operator, which returns the set of tuples that are restricted to $\boldsymbol{E}$. We only consider projection that eliminates duplicates; otherwise it is trivial. Our protocol for projection $\Pi_{\boldsymbol{E}}(R)$ is described in Algorithm 3. It first drops irrelevant attributes $\boldsymbol{F} - \boldsymbol{E}$ from R. To remove duplicates, it then permutes R by $\hat{\boldsymbol{E}}$, checks each pair of consecutive tuples in parallel, and turns the first tuple to dummy if they are the same. Similar to the selection operator, it turns some tuples to dummy, so the ranks should also be updated by Algorithm 2. Meanwhile, we can also drop any rank attribute $\hat{\boldsymbol{E}}'$ if $\boldsymbol{E'} \not\subseteq \boldsymbol{E}$.

[2]The **do in parallel** statement in our algorithm description refers to operations done in parallel, i.e., in the same rounds.

2) Aggregation: Our projection protocol can be extended to support aggregation with "group by" clause. Consider the operation $\Pi_{\boldsymbol{E}}^{\oplus(A)}(R)$, i.e., an aggregation on attribute A with aggregate function $\oplus$ (which can be `count`, `sum`, `max`, or `min`), group by $\boldsymbol{E}$. As projection, we drop irrelevant attributes and ranks (note that A should not be dropped), and then permutes R by $\hat{\boldsymbol{E}}$, so that tuples with same $\boldsymbol{E}$ are consecutive. Note that $\oplus$ is an associative binary operator, so we can compute the aggregation by evaluating the prefix sum circuit segmented by $\boldsymbol{E}$. This way, the last tuple of each segment holds the aggregate for the group, so we set the other tuples to dummy as projection. In addition, we can also support aggregate function `avg` by computing both `sum` and `count` then the division of them [28].

Algorithm 3: Projection protocol $\Pi_{\boldsymbol{E}}$

```
Input: Relation $R(\boldsymbol{F}; \hat{\mathcal{E}})$ with public size $n$
Output: $T(\boldsymbol{E}; \hat{\mathcal{E}}')$
1 Drop attributes in $\boldsymbol{F} - \boldsymbol{E}$ from $R$;
2 $T \leftarrow$ permutation of $R$ by $\hat{\boldsymbol{E}}$;
3 for $i \leftarrow 1$ to $n-1$ do in parallel
4     if $T[i].\boldsymbol{E} = T[i+1].\boldsymbol{E}$ then
5         $T[i] \leftarrow \perp$;
6 $\mathcal{E}' \leftarrow \mathcal{E}_{\subseteq \boldsymbol{E}} = \{\boldsymbol{E}' \in \mathcal{E} \mid \boldsymbol{E}' \subseteq \boldsymbol{E}\}$;
7 Update rank attributes of $T$ by Algorithm 2;
8 return $T$
```

D. Semi-join $\ltimes$

The semi-join operator takes two relations $R(\boldsymbol{F_R}; \hat{\mathcal{E}}_R)$ and $S(\boldsymbol{F}_S; \hat{\mathcal{E}}_S)$ as input, and returns the set of tuples in R that can join with S, i.e., $R \ltimes S = \{t_1 \in R \mid \exists t_2 \in S : t_2.\boldsymbol{E} = t_1.\boldsymbol{E}\}$, where $\boldsymbol{E} = \boldsymbol{F_R} \cap \boldsymbol{F_S}$ is the join key. Similar to the selection operator, the tuples in R that do not join with S are not removed but set to dummy.

The idea is to first permute R by $R.\hat{\boldsymbol{E}}$ and permute S by $S.\hat{\boldsymbol{E}}$. Then for all tuples in S with the same value on $S.\boldsymbol{E}$, we set all but one to dummy. This is done as in Line 3–5 of Algorithm 3. Then we compute the extended `intersection` of $R.\boldsymbol{E}$ and $S.\boldsymbol{E}$. Finally, we set tuples that get a zero payload to dummy and update the ranks by Algorithm 2.

E. Join $\bowtie$

Recall that a join takes two relations $R(\boldsymbol{F}_R; \hat{\mathcal{E}}_R)$ and $S(\boldsymbol{F}_S; \hat{\mathcal{E}}_S)$ as input, where $\hat{\mathcal{E}}_R$ and $\hat{\mathcal{E}}_S$ denote the rank attributes, and returns all combinations of tuples from the two relations such that they have the same values on their join key, i.e., the join result $R \bowtie S$ is $T(\boldsymbol{F}_R \cup \boldsymbol{F}_S; \hat{\mathcal{E}}_R \cup \hat{\mathcal{E}}_S)$, where

$$T = \{(t_R, t_S) \mid t_R \in R, t_S \in S, t_R.\boldsymbol{E} = t_S.\boldsymbol{E}\},$$

and the join key $\boldsymbol{E} = \boldsymbol{F}_R \cap \boldsymbol{F}_S$ is in $\mathcal{E}_R \cap \mathcal{E}_S$. Note that the join result contains all the rank attributes $\hat{\mathcal{E}}_R \cup \hat{\mathcal{E}}_S$, so as to support multi-way joins which are discussed in Section IV.

Assume for simplicity that $|R| = |S| = n$. We also assume for now that the join size $m \geq |R \bowtie S|$ is given. The last $m - |R \bowtie S|$ tuples of the join result T are dummy. The idea follows [8], [9], [29], but due to the rank attributes, we are able to implement it with linear cost. For any tuple $t_R \in R$, note that the number of repetitions of t_R in $R \bowtie S$ is exactly how many times $t_R.\boldsymbol{E}$ appears in $S.\boldsymbol{E}$. Therefore, we first count the frequencies of the values in $S.\boldsymbol{E}$, which can be done by group-by-count $\Pi_{\boldsymbol{E}}^{\texttt{count}}$. Then we permute R by $\boldsymbol{E}$, and attach these frequencies to the corresponding tuples in R, which can be realized by extended `intersection`. Next, we compute the `expansion` of R according to these frequencies. In a symmetric way we also expand S. Then we permute S by some specific permutation indices (i.e., G in Algorithm 4) so as to align each tuple in S with the corresponding joined tuple in R. Finally, zipping the two expanded relations together yields the join results. See Algorithm 4 for details and Figure 1 for an example.

Algorithm 4: Join protocol $\bowtie$

```
Input: Relation $R(\boldsymbol{F_R}; \hat{\mathcal{E}}_R)$ and $S(\boldsymbol{F_S}; \hat{\mathcal{E}}_S)$ each with public
    size $n$
Output: Join result $T(\boldsymbol{F_R} \cup \boldsymbol{F_S}; \hat{\mathcal{E}}_R \cup \hat{\mathcal{E}}_S)$ with public size
    $m$
// Use permutation to sort relations by
    join key in $O(n)$ costs and $O(1)$ rounds.
1 $R \leftarrow$ permutation of $R$ by $R.\boldsymbol{E}$;
2 $S \leftarrow$ permutation of $S$ by $S.\boldsymbol{E}$;
// Calculate the appearance of each tuple
    in $R$ and $S$, and expand the tuple by
    the appearance in $O(n)$ costs and $O(\log n)$
    rounds.
3 $S'(\boldsymbol{E}, \texttt{count}) \leftarrow \Pi_{\boldsymbol{E}}^{\texttt{count}}(S)$;
4 $R.D_S \leftarrow$ extended intersection of $R.\boldsymbol{E}$ and $S'.\boldsymbol{E}$ with
    payload $S'.\texttt{count}$;
5 $R'' \leftarrow$ expansion of $R$ by degree $R.D_S$ with output size
    $m$ (by Algorithm 5);
6 $R'(\boldsymbol{E}, \texttt{count}) \leftarrow \Pi_{\boldsymbol{E}}^{\texttt{count}}(R)$;
7 $S.D_R \leftarrow$ extended intersection of $S.\boldsymbol{E}$ and $R'.\boldsymbol{E}$ with
    payload $R'.\texttt{count}$;
8 $S.D_S \leftarrow$ extended intersection of $S.\boldsymbol{E}$ and $S'.\boldsymbol{E}$ with
    payload $S'.\texttt{count}$;
9 $S.O \leftarrow (1, 2, \dots, n)$;
10 $S.J \leftarrow$ prefix_sum of $(1, \dots, 1)$ segmented by $S.\boldsymbol{E}$;
11 $S'' \leftarrow$ expansion of $S$ by degree $S.D_R$ with output size
    $m$ (by Algorithm 5);
// Rearrange the expanded relation $S$ in
    $O(n)$ costs and $O(\log n)$ rounds.
12 $S''.I \leftarrow$ prefix_sum of $(1, \dots, 1)$ segmented by $S''.O$;
13 Add an attribute $G$ to $S''$;
14 for $i \leftarrow 1$ to $m$ do in parallel
15     $t \leftarrow S''[i]$;
16     $S''[i].G \leftarrow i + (t.I - 1) \cdot t.D_S + t.J - (t.J - 1) \cdot t.D_R - t.I$;
17 $S'' \leftarrow$ permutation of $S''$ by $G$;
// Keep $S''.\hat{\boldsymbol{E}}$ consistent with $R''.\hat{\boldsymbol{E}}$.
18 $S''.\hat{\boldsymbol{E}} \leftarrow (1, \dots, m)$;
19 $T \leftarrow (R''.\boldsymbol{F_R}, S''.\boldsymbol{F_S}; R''.\hat{\mathcal{F}}_R \cup S''.\hat{\mathcal{F}}_S)$;
20 return $T$
```

Example 2. Consider an example of $R(A, B) \bowtie S(B, C)$ with $n = 7$ and $m = 10$ in Figure 1 and transferring the ranks of all three attributes (i.e., $\{(A), (B), (C)\}$).

We handle relation R as the first line of the figure, where we first permute R by (B) and then get the frequencies of corresponding tuples in S, denoted as D_S, which are exactly the number of occurrences in the join result for each tuple in R. Next we compute the `expansion` of R with degree D_S and transfer the valid ranks of $\{(A),(B)\}$. Note that the expanded result are exactly the left part of the final join result in the last figure of the third line.

The next two lines are the way we handle relation S. Permuting is no longer needed since S is already sorted by (B). We compute the frequencies of corresponding tuples in R and S respectively, denoted as D_R and D_S. O, I, J are extra variables for computing specific permutation indices G for alignment. O is the distinct counter of all tuples in S; J is the distinct counter of tuples sharing the same join key (i.e., B) and I is the distinct counter of expanded tuples (i.e., tuples sharing the same O in expanded S). The permutation indices G are computed as line 16, which represents the way to align expanded S with the join result. After the permutation, the expanded S is exactly the right part of the final join result in the last figure of the third line. Zipping the two parts together yields the final join results. $\triangle$

Algorithm 5: Expansion with ranks protocol

```
Input: Relation $R(\boldsymbol{F}, D; \hat{\mathcal{E}})$ with public size $n$
Output: $T(\boldsymbol{F}; \hat{\mathcal{E}})$ with public size $m$
1  Add an attribute $O$ to $R$ with $R[i].O = i$ for all $i \in [n]$;
2  for $\boldsymbol{E} \in \mathcal{E}$ do
3      $R \leftarrow$ permutation of $R$ by $R.\hat{\boldsymbol{E}}$;
4      $(s_1, \ldots, s_n) \leftarrow$ prefix_sum of $R.D$;
5      $R[1].\hat{\boldsymbol{E}} \leftarrow 0$;
6      for $i \leftarrow 2$ to $n$ do in parallel
7          $R[i].\hat{\boldsymbol{E}} \leftarrow s_{i-1}$;
8  $R \leftarrow$ permutation of $R$ by $R.O$;
9  $T(\boldsymbol{F}; \hat{\mathcal{E}}, O) \leftarrow$ expansion of $R$ on $R.D$;
10 $T.P \leftarrow$ prefix_sum of $(1, 1, \ldots, 1)$ segmented by $T.O$;
11 for $\boldsymbol{E} \in \mathcal{E}$ do
12     for $i \leftarrow 1$ to $m$ do in parallel
13         if $T[i] = \perp$ then
14             $T[i].\hat{\boldsymbol{E}} \leftarrow i$;
15         else
16             $T[i].\hat{\boldsymbol{E}} \leftarrow T[i].\hat{\boldsymbol{E}} + T[i].P$;
17 Remove attributes $O, P$ from $T$;
18 return $T$
```

1) Expansion with ranks: In order to support multi-way joins still with linear cost, we must recompute the rank attributes of T. Observing that the only operation in the join algorithm that introduces new tuples is `expansion`, we just need to show how to recompute the ranks after `expansion`.

Let the input be $R(\boldsymbol{F}, D; \hat{\mathcal{E}})$ with size n, where $R.D$ is the degree for expansion, with the promise that $t.D = 0$ if t is dummy. Recall that the output of expansion is a relation $T(\boldsymbol{F}; \hat{\mathcal{E}})$ of size m, where for each $t \in R$, $t.\boldsymbol{F}$ appears $t.D$ times in $T.\boldsymbol{F}$. After the expansion, we need to recompute each rank attribute $\hat{\boldsymbol{E}} \in \hat{\mathcal{E}}$. A natural idea is to permute R by $\hat{\boldsymbol{E}}$, compute the `expansion` of it, and then set $T[i].\hat{\boldsymbol{E}}$ to i for all $i \in [m]$. However, this result is a permutation of the expansion of the original relation, and it is not clear how to permute it to the correct order. Moreover, it does not work if $|\mathcal{E}| \geq 2$, as we could only permute and expand in a specific order.

Our solution is to analyze how the ranks change after expansion: For any $t \in R$, the ranks of the $t.D$ repetitions of t in T should range from $s+1$ to $s+t.D$, where s is the sum of degrees of tuples with ranks less than t in R. Therefore, we first add an attribute O to R to record the initial order of the relation. Then for each $\boldsymbol{E} \in \mathcal{E}$, we permute R by $\hat{\boldsymbol{E}}$, compute the `prefix_sum` of $R.D$, and then update the rank attribute $\hat{\boldsymbol{E}}$. After all the rank attributes are updated, we recover the relation to its original order by permuting it by $R.O$, and then we can compute the `expansion` as usual to get T. The ranks of T in each repetition are then added by its local rank in each segment, which can also be computed by a segmented `prefix_sum`. Recall that the original `expansion` supports the case $m > D_\Sigma$, which results in $m - D_\Sigma$ dummy tuples at the end of T, and their ranks are not computed correctly. We update their ranks to the current orders in T. See Algorithm 5 and Figure 2 for details. The cost and the number of rounds are $O(n+m)$ and $O(\log(n+m))$ respectively.

IV. Secure Query Processing

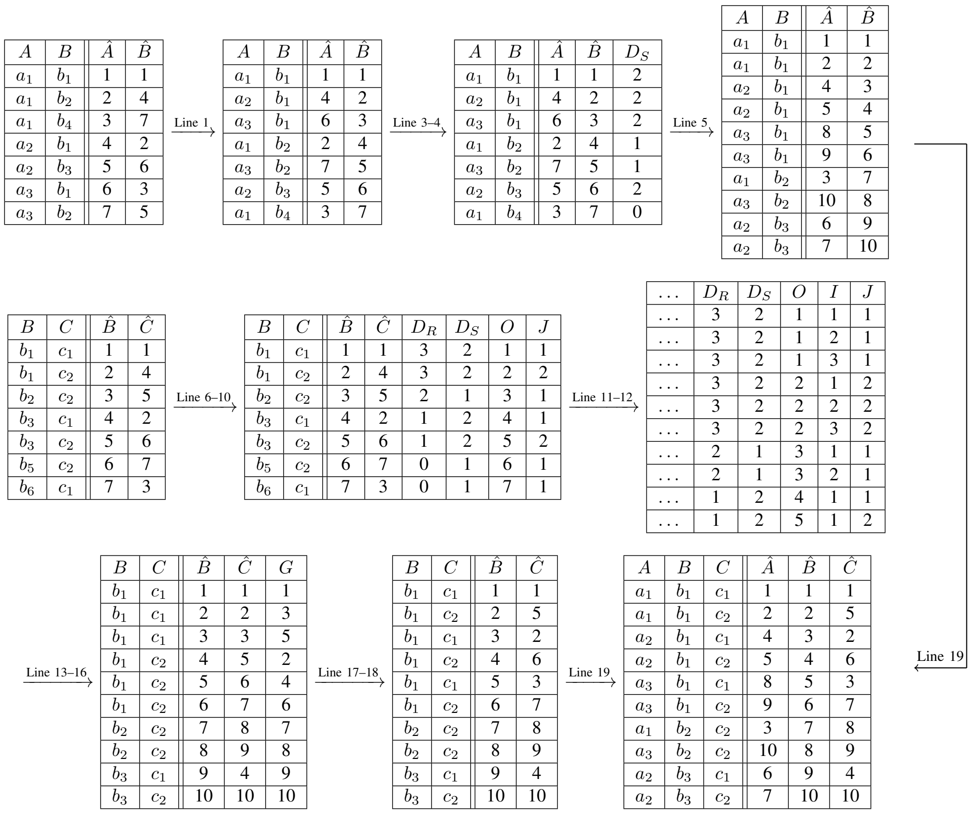
In this section, we show how our protocols for relational operators can be composed to answer *free-connex queries*.

In the original definition of a free-connex query, the projection operator $\Pi_{\boldsymbol{O}}$ can also be replaced by some aggregation operator $\Pi^{\oplus}_{\boldsymbol{O}}$, which we discuss in Section IV-B.

A. Protocol for Free-connex Queries

Assume for simplicity that the size of each relation $|R_i| = n$ and the output size $|\mathcal{Q}| = m$ is given. Because our protocols take one or two relations with ranks as input and outputs a relation with ranks, they can be used with any query plan $\mathcal{P}$ consisting of these operators, with cost proportional to the input and output size of each operator. For selection, projection, aggregation, and semi-join, the output size is always equal to the input size (recall that we pad the output with dummy tuples for security reasons). Our join protocol, however, needs an upper bound m on the join size, and its cost is $O(n+m)$. Thus, to guarantee an overall linear cost, we need the query plan $\mathcal{P}$ provided by Yannakakis algorithm [1] and its generalized version [2], [3], [30] which ensures that all the intermediate join sizes are bounded by m. The query plan $\mathcal{P}$ consists of operators in $\{\sigma, \Pi, \ltimes, \bowtie\}$. Specifically, the Yannakakis algorithm works as follows:

1) *Reduce.* The *reduce* step works by visiting the free-connex join tree $\mathcal{T}$ in a bottom-up order. For each node $R(\boldsymbol{F_R}) \in \mathcal{T}$ and its parent node $P(\boldsymbol{F_P})$, let $\boldsymbol{F_{R'}} = (\boldsymbol{O} \cup \boldsymbol{F_P}) \cap \boldsymbol{F_R}$. We update node R to $R' \leftarrow \Pi_{\boldsymbol{F_{R'}}}(R)$. If $\boldsymbol{F_{R'}} \subseteq \boldsymbol{F_P}$, we update $P \leftarrow P \ltimes R'$, and then remove R' from $\mathcal{T}$; otherwise, stop going upward. This pass terminates when either the root is the only node in $\mathcal{T}$,



A	B	$\hat{A}$	$\hat{B}$
a_1	b_1	1	1
a_1	b_2	2	4
a_1	b_4	3	7
a_2	b_1	4	2
a_2	b_3	5	6
a_3	b_1	6	3
a_3	b_2	7	5

$\xrightarrow{\text{Line 1}}$

A	B	$\hat{A}$	$\hat{B}$
a_1	b_1	1	1
a_2	b_1	4	2
a_3	b_1	6	3
a_1	b_2	2	4
a_3	b_2	7	5
a_2	b_3	5	6
a_1	b_4	3	7

$\xrightarrow{\text{Line 3–4}}$

A	B	$\hat{A}$	$\hat{B}$	D_S
a_1	b_1	1	1	2
a_2	b_1	4	2	2
a_3	b_1	6	3	2
a_1	b_2	2	4	1
a_3	b_2	7	5	1
a_2	b_3	5	6	2
a_1	b_4	3	7	0

$\xrightarrow{\text{Line 5}}$

A	B	$\hat{A}$	$\hat{B}$
a_1	b_1	1	1
a_1	b_1	2	2
a_2	b_1	4	3
a_2	b_1	5	4
a_3	b_1	8	5
a_3	b_1	9	6
a_1	b_2	3	7
a_3	b_2	10	8
a_2	b_3	6	9
a_2	b_3	7	10

B	C	$\hat{B}$	$\hat{C}$
b_1	c_1	1	1
b_1	c_2	2	4
b_2	c_2	3	5
b_3	c_1	4	2
b_3	c_2	5	6
b_5	c_2	6	7
b_6	c_1	7	3

$\xrightarrow{\text{Line 6–10}}$

B	C	$\hat{B}$	$\hat{C}$	D_R	D_S	O	J
b_1	c_1	1	1	3	2	1	1
b_1	c_2	2	4	3	2	2	2
b_2	c_2	3	5	2	1	3	1
b_3	c_1	4	2	1	2	4	1
b_3	c_2	5	6	1	2	5	2
b_5	c_2	6	7	0	1	6	1
b_6	c_1	7	3	0	1	7	1

$\xrightarrow{\text{Line 11–12}}$

...	D_R	D_S	O	I	J
...	3	2	1	1	1
...	3	2	1	2	1
...	3	2	1	3	1
...	3	2	2	1	2
...	3	2	2	2	2
...	3	2	2	3	2
...	2	1	3	1	1
...	2	1	3	2	1
...	1	2	4	1	1
...	1	2	5	1	2

$\xrightarrow{\text{Line 13–16}}$

B	C	$\hat{B}$	$\hat{C}$	G
b_1	c_1	1	1	1
b_1	c_1	2	2	3
b_1	c_1	3	3	5
b_1	c_2	4	5	2
b_1	c_2	5	6	4
b_1	c_2	6	7	6
b_2	c_2	7	8	7
b_2	c_2	8	9	8
b_3	c_1	9	4	9
b_3	c_2	10	10	10

$\xrightarrow{\text{Line 17–18}}$

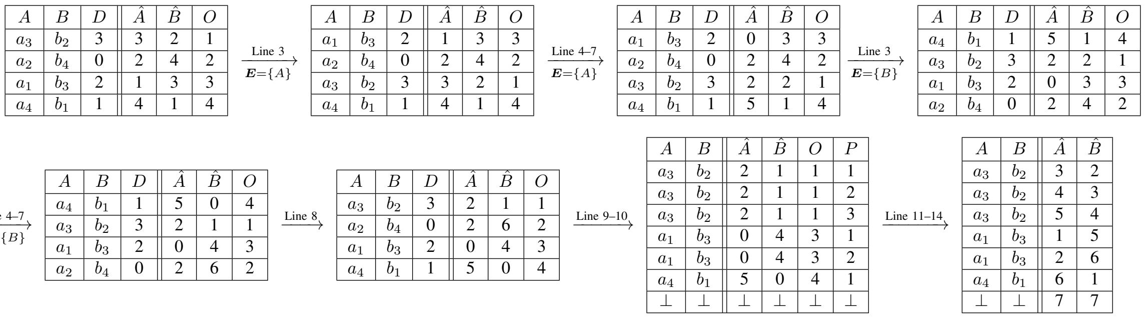
B	C	$\hat{B}$	$\hat{C}$
b_1	c_1	1	1
b_1	c_2	2	5
b_1	c_1	3	2
b_1	c_2	4	6
b_1	c_1	5	3
b_1	c_2	6	7
b_2	c_2	7	8
b_2	c_2	8	9
b_3	c_1	9	4
b_3	c_2	10	10

$\xrightarrow{\text{Line 19}}$

A	B	C	$\hat{A}$	$\hat{B}$	$\hat{C}$
a_1	b_1	c_1	1	1	1
a_1	b_1	c_2	2	2	5
a_2	b_1	c_1	4	3	2
a_2	b_1	c_2	5	4	6
a_3	b_1	c_1	8	5	3
a_3	b_1	c_2	9	6	7
a_1	b_2	c_2	3	7	8
a_3	b_2	c_2	10	8	9
a_2	b_3	c_1	6	9	4
a_2	b_3	c_2	7	10	10

$\xleftarrow{\text{Line 19}}$

Fig. 1: An example of join Algorithm 4 with $n = 7$, $m = 10$, $\boldsymbol{F_R} = \{A, B\}$, $\mathcal{E}_R = \{(A), (B)\}$, $\boldsymbol{F_S} = \{B, C\}$, and $\mathcal{E}_R = \{(B), (C)\}$



A	B	D	$\hat{A}$	$\hat{B}$	O
a_3	b_2	3	3	2	1
a_2	b_4	0	2	4	2
a_1	b_3	2	1	3	3
a_4	b_1	1	4	1	4

$\xrightarrow[\boldsymbol{E}=\{A\}]{\text{Line 3}}$

A	B	D	$\hat{A}$	$\hat{B}$	O
a_1	b_3	2	1	3	3
a_2	b_4	0	2	4	2
a_3	b_2	3	3	2	1
a_4	b_1	1	4	1	4

$\xrightarrow[\boldsymbol{E}=\{A\}]{\text{Line 4–7}}$

A	B	D	$\hat{A}$	$\hat{B}$	O
a_1	b_3	2	0	3	3
a_2	b_4	0	2	4	2
a_3	b_2	3	2	2	1
a_4	b_1	1	5	1	4

$\xrightarrow[\boldsymbol{E}=\{B\}]{\text{Line 3}}$

A	B	D	$\hat{A}$	$\hat{B}$	O
a_4	b_1	1	5	1	4
a_3	b_2	3	2	2	1
a_1	b_3	2	0	3	3
a_2	b_4	0	2	4	2

$\xrightarrow[\boldsymbol{E}=\{B\}]{\text{Line 4–7}}$

A	B	D	$\hat{A}$	$\hat{B}$	O
a_4	b_1	1	5	0	4
a_3	b_2	3	2	1	1
a_1	b_3	2	0	4	3
a_2	b_4	0	2	6	2

$\xrightarrow{\text{Line 8}}$

A	B	D	$\hat{A}$	$\hat{B}$	O
a_3	b_2	3	2	1	1
a_2	b_4	0	2	6	2
a_1	b_3	2	0	4	3
a_4	b_1	1	5	0	4

$\xrightarrow{\text{Line 9–10}}$

A	B	$\hat{A}$	$\hat{B}$	O	P
a_3	b_2	2	1	1	1
a_3	b_2	2	1	1	2
a_3	b_2	2	1	1	3
a_1	b_3	0	4	3	1
a_1	b_3	0	4	3	2
a_4	b_1	5	0	4	1
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\xrightarrow{\text{Line 11–14}}$

A	B	$\hat{A}$	$\hat{B}$
a_3	b_2	3	2
a_3	b_2	4	3
a_3	b_2	5	4
a_1	b_3	1	5
a_1	b_3	2	6
a_4	b_1	6	1
$\perp$	$\perp$	7	7

Fig. 2: An example of Algorithm 5 with $n = 4$, $m = 7$, $\boldsymbol{F} = \{A, B\}$, and $\mathcal{E} = \{(A), (B)\}$

or all attributes in $\mathcal{T}$ are output attributes. In the former case, the algorithm terminates, and the relation at the root node is exactly the result of $\mathcal{Q}$. In the latter case, since all attributes are in $\boldsymbol{O}$, the query degenerates to a join query. It then goes to the next step.

2) *Semi-joins.* Next, the algorithm uses two passes of *semi-joins* on the join tree to remove all the *dangling tuples*, i.e., tuples that do not contribute to the join result. Specifically, we first use a bottom-up pass to check each pair of relation R and its parent P and update $P \leftarrow P \ltimes R$; then use a top-down pass to check each pair of relation R and its parent P and update $R \leftarrow R \ltimes P$.
3) *Joins.* Finally, compute the join query in a bottom-up pass: While there is more than one node in $\mathcal{T}$, pick a leaf relation R and its parent P, update $P \leftarrow P \bowtie R$ with join size upper bound m, and remove R from tree. The process terminates when there remains the only node in $\mathcal{T}$, which is the query result.

By executing the query plan above, we derive the following main theorem of this paper:

Theorem IV.1. *Given a free-connex query $\mathcal{Q}$ with input relations $R_1(\boldsymbol{F}_1), \ldots, R_k(\boldsymbol{F}_k)$, there exists $\{\mathcal{E}_i \subseteq 2^{\boldsymbol{F}_i}\}$ such that if for each i, the rank $\hat{\mathcal{E}}_i$ in R_i is available, then there exists a 3PC protocol that computes $\mathcal{Q}$ in linear cost.*

Proof. Let $\mathcal{T}$ be a free-connex join tree of $\mathcal{Q}$. For any node $R \in \mathcal{T}$, let P be its parent node and $\mathcal{C}$ be the set of its children nodes. We define the rank attributes of R as

$$\mathcal{E}_R = \{(\boldsymbol{O} \cup \boldsymbol{F}_P) \cap \boldsymbol{F}_R, \boldsymbol{F}_P \cap \boldsymbol{F}_R\} \cup \{\boldsymbol{F}_C \cap \boldsymbol{F}_R\}_{C \in \mathcal{C}}.$$

We then prove that with ranks $\{\hat{\mathcal{E}}_R\}$, we can solve $\mathcal{Q}$ with $O(n+m)$ cost, where n is the total input size and m is the output size.

First we go through the reduce step. For any relation R with its parent P, the projection $R' = \Pi_{\boldsymbol{F_{R'}}}(R)$ incurs $O(n)$ cost because $\boldsymbol{F_{R'}} = (\boldsymbol{O} \cup \boldsymbol{F_P}) \cap \boldsymbol{F_R} \in \mathcal{E}_R$. For the semi-join $P \ltimes R'$, note that $\boldsymbol{F_P} \cap \boldsymbol{F_{R'}} = \boldsymbol{F_P} \cap \boldsymbol{F_R}$, which also belongs to both $\mathcal{E}_R$ and $\mathcal{E}_P$, and hence incurs $O(n)$ cost.

Then we go through the *semi-joins* step. Note that the reduce step does not make new parent-child pair, because once a node is removed, all of its children are already removed. Hence, any semi-join operation is either in the form $P \ltimes R'$ or $R' \ltimes P$. As discussed above, it incurs $O(n)$ cost.

Finally we go through the *joins* step. It has been shown by Yannakakis algorithm that for each join operation, its output size is bounded by m since dangling tuples have been removed, hence both the total input size and the output size of each join operation is $O(n+m)$. For any relation R' with its parent P, it is obvious that the set of attributes remain unchanged during previous steps. Suppose the original node of R' is R, then $\boldsymbol{F_R}$ is changed to $\boldsymbol{F_{R'}}$ by two cases: (1) a projection operation on R in the reduce step, which we have proved not to influence the intersection $\boldsymbol{F_P} \cap \boldsymbol{F_R}$, and (2) the join of R with all of its descendants in the *joins* step. We then prove that (2) also does not change the intersection $\boldsymbol{F_P} \cap \boldsymbol{F_R}$, so $\boldsymbol{F_P} \cap \boldsymbol{F_{R'}} = \boldsymbol{F_P} \cap \boldsymbol{F_R}$. If this is not the case, then there exists an attribute A that $A \in \boldsymbol{F_P}$, $A \notin \boldsymbol{F_R}$, and $A \in \boldsymbol{F_{R'}}$. This suggests that $A \in F_D$ for some descendant relation D of R in the original free-connex join tree, which is contradicted to the definition of free-connex join tree because the nodes that contain A are not connected. Therefore, we conclude that $\boldsymbol{F_P} \cap \boldsymbol{F_{R'}}$ belongs to both $\mathcal{E}_R$ and $\mathcal{E}_P$, and hence incurs $O(n+m)$ cost. □

Example 3. The following query is free-connex:

$$\Pi_{A,B,C,D,E}(R_1(A,B) \bowtie R_2(A,D,E) \\ \bowtie R_3(B,C,F) \bowtie R_4(C,F,G)).$$

One possible join tree is shown in the upper left of Figure 3. It induces the query plan as follows:

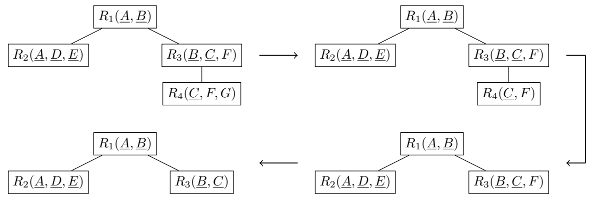
1) *Reduce.* $R_4(C,F) \leftarrow \Pi_{C,F}(R_4)$; $R_3(B,C,F) \leftarrow R_3 \bowtie R_4$; $R_3(B,C) \leftarrow \Pi_{B,C}(R_3)$. Figure 3 shows how the join tree changes during this step.
2) *Semi-joins.* $R_1 \leftarrow R_1 \ltimes R_2$; $R_1 \leftarrow R_1 \ltimes R_3$; $R_2 \leftarrow R_2 \ltimes R_1$; $R_3 \leftarrow R_3 \ltimes R_1$.
3) *Joins.* $R_1(A,B,D,E) \leftarrow R_1 \bowtie R_2$; $R_1(A,B,C,D,E) \leftarrow R_1 \bowtie R_3$; R_1 is the output. △

B. Queries with Aggregation

A free-connex query with aggregation is in the form

$$\mathcal{Q} = \Pi^{\oplus}_{\boldsymbol{O}}\big(\sigma_{\gamma_1}(R_1(\boldsymbol{F}_1)) \bowtie \cdots \bowtie \sigma_{\gamma_k}(R_k(\boldsymbol{F}_k))\big),$$

where the aggregate function $\oplus$ is applied on the *annotation* of its input relation. We follow the same terminology from [2], [13] to define an *annotation*. Let $(\mathcal{S}, \oplus, \otimes)$ be a communicative semiring. Each tuple t is associated with an annotation $v \in \mathcal{S}$. If an annotation is not needed for a tuple, we set it to the $\otimes$-identity of the semiring. We conceptually add the annotations as a special attribute V, so the servers will store $R = R(\boldsymbol{F}; V; \hat{\mathcal{E}})$ in secret-shared form. In the implementation, this is not needed if V is one of the existing attributes of R, or can be computed on-the-fly when V is a function of the attributes, e.g., `price * quantity`.



$R_1(\underline{A},\underline{B})$
$R_2(\underline{A},\underline{D},\underline{E})$
$R_3(\underline{B},\underline{C},F)$
$R_4(\underline{C},F,G)$
$R_4(\underline{C},F)$
$R_3(\underline{B},\underline{C})$

Fig. 3: Examples of the *reduce* step of the Yannakakis algorithm with output attributes underlined

Next we specify how the annotations propagate through join and aggregation. The join $R \bowtie S$ of two annotated relations R, S also returns an annotated relation, where the tuple joined by $t_R \in R$ and $t_S \in S$ has the annotation $t_R.V \otimes t_S.V$, which can be directly obtained in linear cost. The aggregation operator, $\Pi^{\oplus}_{\boldsymbol{O}}$, over an annotated relation R, computes the $\oplus$-aggregate of each group of tuples grouped by $\boldsymbol{O}$. Since we have proposed a protocol for an aggregation operator with linear cost, it is direct to replace each projection operator $\pi_{\boldsymbol{E}}$ in the *reduce* step with aggregation operator $\pi^{\oplus}_{\boldsymbol{E}}$ to support free-connex query with aggregation with still linear online cost.

C. Compute Output Size

While the Yannakakis algorithm guarantees that any intermediate join size is bounded by m, the final query output size, we still need to compute it, which is essential by our 3PC join protocol. Recall the three steps of the Yannakakis algorithm. Observe that the first two steps (*reduce* and *semi-joins*) do not require m, which we still run as before. After these two steps, we obtain a join query $\mathcal{J}$ without dangling tuples, and its output size is the same as that of the original query $\mathcal{Q}$. To compute the output size of $\mathcal{J}$, we turn it into another free-connex query with aggregation $\mathcal{Q}' = \Pi^{\mathtt{sum}}_{\varnothing}(\mathcal{J})$, where the annotations of all tuples of input relations are set to 1, and we use the semiring $(\mathbb{Z}, +, \times)$, i.e., the ring of integers. Since $\mathcal{Q}'$ has no output attributes, on this query the algorithm will terminate in step (1), which returns $m = |\mathcal{Q}|$. Then we can continue with step (3) on $\mathcal{Q}$.

Example 4. Consider the free-connex query in Example 3. After the first two steps, we have $\mathcal{J} = R_1(A,B) \bowtie R_2(A,D,E) \bowtie R_3(B,C)$, and its join tree is the last tree in Figure 3. We set annotations of all tuples of R_1, R_2, R_3 to 1. Then we run just the *reduce* step on $\mathcal{Q}'$: $R_2(A) \leftarrow \Pi^{\mathtt{sum}}_{A}(R_2)$; $R_1(A,B) \leftarrow R_1 \bowtie R_2$; $R_3(B) \leftarrow \Pi^{\mathtt{sum}}_{B}(R_3)$; $R_1(A,B) \leftarrow R_1 \bowtie R_3$; $m \leftarrow \Pi^{\mathtt{sum}}_{\varnothing}(R_1)$. △

V. Security Analysis

We analyze the security of our protocol in the 3PC model in the presence of a semi-honest adversary. Given a free-connex query plan, the protocol is a sequential composition of relational operators, which are further sequentially built from primitive protocols, including circuit-based computations, permutations, expansions, and intersections. Our protocol has the following properties: (1) each primitive has been shown to be secure in Section II-E (i.e., satisfying the real-ideal paradigm), (2) all intermediate relations are stored in secret-shared form, and (3) the intermediate sizes depend only on input size n and query result size m. Therefore, its security is guaranteed by the composition theorem [27].

VI. System Implementation

A. Implementation Overview

We have built a system prototype SLOOP on top of ABY3 [23], [31] for evaluating any free-connex query under the 3PC model. ABY3 is secure against semi-honest adversaries and already provides the implementation of some three-party primitives like circuit-based computations and intersection. For those that are not, such as prefix sum circuit, permutation, compaction, and expansion, we have done our own implementation in C++. The default bit length of each attribute is 64. Security parameter is set to $\kappa = 128$.

SLOOP works as follows. In the preprocessing stage, the servers generate useful ranks on the input relations (Section III-A). During the online execution, our system first select one free-connex join tree when the query is given. Then the servers execute the query by the protocol based on the selected free-connex join tree. After the query processing is done, the servers transmit the secret shares of the query result to the client, who can then reconstruct the final plaintext query result.

B. Optimization

We also proposed some optimization techniques in SLOOP, which do not lead to theoretical improvements in data complexity, but significantly improve the performance by reducing the hidden constant factors.

1) Preprocessing Optimization: In the preprocessing stage for generating ranks, we observe that it is not necessary to sort the entire relation to generate a single rank. Instead, we only focus on sorting the key. Specifically, to generate $\hat{\boldsymbol{E}}$ in relation $R(\boldsymbol{F})$, we first construct a subrelation $R'(\boldsymbol{E}, C)$ where $t'[i].\boldsymbol{E} \leftarrow t[i].\boldsymbol{E}$ and $t'[i].C \leftarrow i$, where $t[i]$ and $t'[i]$ are the i-th tuple of R and R' respectively. We then sort $R'(\boldsymbol{E}, C)$ by $\boldsymbol{E}$, add a new attribute $\hat{\boldsymbol{E}}$ such that $t'[i].\hat{\boldsymbol{E}} \leftarrow i$, and drop attributes $\boldsymbol{E}$ from R'. Finally, we permute $R'(C; \hat{\boldsymbol{E}})$ on C and append $\hat{\boldsymbol{E}}$ to the original relation R. Compared with the naive solution that simply sorts the relation by $\boldsymbol{E}$, the hidden constant of the sorting primitive introduced by the size of sorting key is therefore is reduced from $|\boldsymbol{F}|$ to 2.

2) Query Plan Optimization: Unlike plaintext database optimizer, the costs of operators in 3PC are all data-independent, so an accurate cost model in MPC plays a more important role. SLOOP adopts a simple cost model by taking both data complexity and the number of attributes (including the rank attributes) of each relation into consideration. Since there can be multiple free-connex join trees for a free-connex query $\mathcal{Q}$, and different free-connex join trees lead to different query plans, SLOOP can therefore use its cost model to choose the best query plan. We introduce the cost model of SLOOP as follows. For any relation $R(\boldsymbol{F}; \hat{\mathcal{E}})$, $|\boldsymbol{F}|$ is the number of original attributes and $|\mathcal{E}|$ is the number of rank attributes. The cost of the selection operator comes from two steps. The first step is to apply the predicates, which is at most linear to $|\boldsymbol{F}|$. The second step is applying Algorithm 2 to update the rank attributes. We note that for each $\boldsymbol{E} \in \mathcal{E}$, updating $\hat{\boldsymbol{E}}$ actually does not require permuting the whole relation. Instead, we first compute the `permutation` of (O, Z) by $\hat{\boldsymbol{E}}$, where $O = [n]$ and Z is the dummy marker. Then we compute the `compaction` of it, set the i-th element of $\hat{\boldsymbol{E}}$ to i for each $i \in [n]$, and permute it to the original order by computing the `permutation` of $\hat{\boldsymbol{E}}$ by O. Since these permutations and the compaction are related to only constant number of attributes, the total cost of Algorithm 2 is linear to $|\mathcal{E}|$. Therefore, the total cost of the selection operator is at most linear to $|\boldsymbol{F}| + |\mathcal{E}|$. The cost of projection/aggregation operator is also at most linear to $|\boldsymbol{F}| + |\mathcal{E}|$ due to the same reason. This trick also works for updating the rank attributes in expansion. Specifically, for each rank attribute $\boldsymbol{E} \in \mathcal{E}$, we update $\hat{\boldsymbol{E}}$ in Line 3–7 in Algorithm 5. The permutation actually does not need to be performed over the whole relation R, but only over O and D, so the total cost of updating the rank attributes in expansion is linear to $|\mathcal{E}|$. The total cost of Algorithm 5 is therefore at most linear to $|\boldsymbol{F}| + |\mathcal{E}|$.

The semi-join operator has cost linear to $|\boldsymbol{F_R}| + |\mathcal{E}_R|$, where $|\boldsymbol{F_R}|$ is for the extended `intersection` on the common attributes of R and S, and $|\mathcal{E}_R|$ is for updating rank attributes by Algorithm 2. For the join operator, the cost is dominated by computing the `expansion` of R and S, so the total cost is linear to $|\boldsymbol{F_R}| + |\mathcal{E}_R| + |\boldsymbol{F_S}| + |\mathcal{E}_S|$.

In conclusion, all operators have cost at most linear to the total number of attributes (including the rank attributes) of the input relations (despite semi-join which is linear to the first relation only). We use this value multiplied by the data complexity as the cost estimation of an operator. The cost of a join tree is therefore the total costs of the operators in the query plan it induces.

Example 5. Recall the free-connex query in Example 3. Consider a new free-connex join tree of this query, in which we put R_2 as the root, i.e., the join tree becomes a line R_2–R_1–R_3–R_4. To compare the two plans, we measure their total costs in the *semi-joins* step and *final join* step, because they have the same query plan in the *reduce* step. The details are shown in Table IV, which suggest we should choose the new free-connex join tree. It can be verified that the cost of this new plan $(15n + 15m)$ is less than the plan in Example 3 $(15n + 16m)$, so we should choose the new free-connex join tree. △

Operation	Cost
$R_1(A,B;\hat{A},\hat{B}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \ltimes R_2(A,D,E;\hat{A})$	$4n$
$R_1(A,B;\hat{A},\hat{B}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \ltimes R_3(B,C;\hat{B})$	$4n$
$R_2(A,D,E;\hat{A}) \leftarrow R_2(A,D,E;\hat{A}) \ltimes R_1(A,B;\hat{A},\hat{B})$	$4n$
$R_3(B,C;\hat{B}) \leftarrow R_3(B,C;\hat{B}) \ltimes R_1(A,B;\hat{A},\hat{B})$	$3n$
$R_1(A,B,D,E;\hat{B}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \bowtie R_2(A,D,E;\hat{A})$	$8m$
$R_1(A,B,C,D,E) \leftarrow R_1(A,B,D,E;\hat{B}) \bowtie R_3(B,C;\hat{B})$	$8m$
Total cost of old join tree	$15n + 16m$

Operation	Cost
$R_1(A,B;\hat{A},\hat{B}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \ltimes R_3(B,C;\hat{B})$	$4n$
$R_2(A,D,E;\hat{A}) \leftarrow R_2(A,D,E;\hat{A}) \ltimes R_1(A,B;\hat{A},\hat{B})$	$4n$
$R_1(A,B;\hat{A},\hat{B}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \ltimes R_2(A,D,E;\hat{A})$	$4n$
$R_3(B,C;\hat{B}) \leftarrow R_3(B,C;\hat{B}) \ltimes R_1(A,B;\hat{A},\hat{B})$	$3n$
$R_1(A,B,C;\hat{A}) \leftarrow R_1(A,B;\hat{A},\hat{B}) \bowtie R_3(B,C;\hat{B})$	$7m$
$R_2(A,B,C,D,E) \leftarrow R_2(A,D,E;\hat{A}) \bowtie R_1(A,B,C;\hat{A})$	$8m$
Total cost of new join tree	$15n + 15m$

TABLE IV: The costs of two join trees

VII. Experiments

A. Experiment Setup

All experiments are measured on three servers from different cloud service providers located in the same region. The latency and bandwidth between the servers are 10ms and 1Gb/s, respectively. We report the longest running time and the maximum communication costs (including data sent and received) among the three servers. All reported results are the average over 10 runs.

We compare SLOOP with plaintext algorithm, OptScape, and SECYAN [13]. The running time of plaintext algorithm is measured by PostgreSQL, and the communication cost refers to the total input and output size. OptScape is an optimized version of Scape, where we replace their $O(n \log^2 n)$ bitonic sorter with the $O(n \log n)$ 3PC sorting protocol [20]. Note that our experimental results on $n = 2^{16}$ tuples indicate that OptScape achieves a speedup of 10x over Scape [9] and 5x over SSJ [8] and SSA [7], thus it serves as the state of the art baseline under 3PC model. The primitives of OptScape are the version that defend against semi-honest adversary, the same as SLOOP. We also compare with SECYAN because it is the state-of-the-art protocol that also supports free-connex queries. However, its security model is incomparable to SLOOP's since it is under 2PC (stronger) but with extra requirements of plaintext input and output (weaker). We also include SLOOP (w/o ranks), representing the scenario where SLOOP prepares no ranks before any query. It includes generation costs (limited to only the required ranks for the query) and the query costs.

B. TPC-H Queries

The first set of queries we tested are summarized below.

- **Q3 / Q10 / Q18**: Three standard free-connex queries from the TPC-H benchmark that are also tested in [13].
- **Q11**: Another free-connex query from TPC-H.
- **Q3F / Q5F**: Multi-way joins of all involved relations in TPC-H Q3 (three relations) / Q5 (six relations) without any selection or projection.

Note that for queries we tested, the intermediate size of any join operator always equals to the size of one of the input relation, which is public. Therefore, the two-way join protocol of OptScape can be directly composed without breaching security guarantee and its total cost is $O(n \log n + m \log m)$. The TPC-H datasets generated by its database population program with scales varying from 0.001 to 1.

The preprocessing and storage costs of all 13 ranks for these queries are in Table V. Not all ranks are used by each query, and some ranks are reused by multiple queries. For example, **Q3**, **Q3F**, **Q10**, and **Q18** use the same 4 ranks, while **Q11** uses totally different 4 ranks.

TABLE V: Preprocessing Costs of Generating All Ranks

Scale	0.001	0.01	0.1	1
Time (s)	3.23	10.66	96.91	1166.99
Comm (MB)	55.64	671.14	8408.92	104775.39
Storage (KB)	133	1325	13228	132207

The experimental results of SLOOP (preprocessing costs excluded) and baselines are presented in Table VI. We see that the running time of our system is consistently 3x–8x lower than that of OptScape, and the communication cost is 4x–8x lower. More importantly, our protocol demonstrates linear growth rates in communication cost, as predicted by the theory. In contrast, OptScape exhibits a logarithmic factor growth, so the gap between them will grow wider as the data size further increases. SECYAN also demonstrates a logarithmic factor growth: 2x–3x larger than ours. Its time cost is similar to ours when the data size is small, but differs a lot when the data size increases due to its extra logarithm factor. With our linear-cost protocol, the running time of MPC query processing is around 50x that of plaintext, with a communication cost that is <100x that of the input and output size. Even though SLOOP (w/o ranks) does not precompute any ranks in advance, its performance still surpasses OptScape's because it reduces repeated sorting. However, it no longer exhibits linear growth rates in communication cost.

C. Graph Queries

Next, we tested queries on graph data. We used the bitcoin-alpha dataset (called "bitcoin") [32], [33], which is a directed graph storing who-trusts-whom network of people. There are 24186 edges in the graph; each tuple represents an edge, containing four attributes: source (`src`), target (`tgt`), `rating` and `time`. We test the following query with tuned parameter k for selection conditions:

```
SELECT b1.src, b1.tgt, b2.tgt, b3.tgt
  FROM bitcoin AS b1
  JOIN bitcoin AS b2 ON b1.tgt = b2.src
  JOIN bitcoin AS b3 ON b2.tgt = b3.src
 WHERE b1.rating >= k
   AND b2.rating >= k
   AND b3.rating >= k;
```

We tried this query with parameters $k = 6, 5, 4, 3$ which lead to different output sizes m. Unlike TPC-H queries, the intermediate join size should be kept secret and the inputs

TABLE VI: Costs of different protocols for TPC-H queries

Time (s)

Query	**Q3**				**Q10**				**Q18**			
Scale	0.001	0.01	0.1	1	0.001	0.01	0.1	1	0.001	0.01	0.1	1
Plaintext	0.003	0.02	0.30	1.78	0.01	0.18	1.08	4.02	0.01	0.09	0.99	3.05
OptScape	1.59	10.11	104.79	1256.49	2.43	10.21	106.15	1245.78	2.48	10.22	104.77	1254.26
SECYAN	0.46	3.68	36.90	377.13	0.44	3.49	34.13	337.69	0.76	5.93	57.65	548.87
SLOOP	0.41	1.53	14.25	166.80	0.46	1.47	13.34	154.70	0.44	1.52	14.47	168.42
SLOOP (w/o ranks)	1.58	6.87	65.92	790.94	1.90	6.50	69.53	779.72	1.87	6.62	76.78	792.42

Query	**Q11**				**Q3F**				**Q5F**			
Scale	0.001	0.01	0.1	1	0.001	0.01	0.1	1	0.001	0.01	0.1	1
Plaintext	0.0005	0.004	0.02	0.28	0.02	0.22	1.92	19.2	0.019	0.21	1.74	18.19
OptScape	1.40	2.81	15.27	147.56	3.26	13.56	132.63	1643.88	6.13	24.67	232.98	2898.92
SLOOP	0.46	0.77	3.61	35.92	1.16	4.64	41.34	477.47	2.29	7.69	72.21	803.07
SLOOP (w/o ranks)	0.88	1.47	5.24	45.04	2.61	9.97	93.03	1101.88	4.87	16.71	160.28	1866.87

Communication (MB)

Query	**Q3**				**Q10**				**Q18**			
Scale	0.001	0.01	0.1	1	0.001	0.01	0.1	1	0.001	0.01	0.1	1
Plaintext	0.23	2.32	23.13	231.20	0.13	1.32	13.17	131.63	0.14	1.39	13.86	138.49
OptScape	59.55	752.11	9572.57	113636.14	59.07	756.63	9584.31	113562.32	60.93	753.15	9487.35	114986.29
SECYAN	15.95	185.50	2118.29	24004.68	12.33	142.82	1638.70	18607.46	23.51	266.82	2999.72	33510.06
SLOOP	7.70	76.82	767.77	7682.20	6.67	66.55	665.07	6655.33	7.36	73.44	734.02	7344.96
SLOOP (w/o ranks)	37.38	427.91	5303.18	63780.08	36.35	427.64	5200.48	62753.21	37.04	434.53	5269.43	63442.84

Query	**Q11**				**Q3F**				**Q5F**			
Scale	0.001	0.01	0.1	1	0.001	0.01	0.1	1	0.001	0.01	0.1	1
Plaintext	0.02	0.19	1.89	19.22	0.41	4.15	41.46	414.35	0.44	4.38	43.76	437.64
OptScape	6.97	90.12	1192.78	12966.69	78.27	921.01	10886.30	129971.91	129.83	1622.45	19006.96	226924.04
SLOOP	1.99	19.26	192.17	1921.44	20.88	208.21	2078.34	20774.26	34.42	341.93	3411.10	34092.46
SLOOP (w/o ranks)	2.36	23.45	255.43	2652	50.56	569.30	6613.75	76872.14	85.99	953.99	11110.04	129547.64

contain dummy tuples, hence OptScape does not support this query, and we do not compare with it. We also do not compare with SECYAN because it does the joins by one of the party locally, which degenerates to the plaintext algorithm. We tried to compare with Secrecy [11] and QCircuit [12], but neither finished the query in 10 hours. To demonstrate how much our proposed ranks contribute to the efficiency speed-up, we compared with SLOOP$^-$, the variant of SLOOP that ignores all the ranks. SLOOP$^-$ works the same as SLOOP for free-connex queries (i.e., protecting intermediate join size) but replaces all permutation primitives by sorting, resulting in a total cost of $O(n \log n + m \log m)$.

The results are in Table VII. Based on the schema, we prepare ranks for `src` and `tgt` respectively. The preprocessing costs of SLOOP are 1.48s and 95.42MB, which is negligible compared with online costs, and the storage cost is 189KB. SLOOP (w/o ranks) has similar performance to SLOOP, while saving one sorting step of size m compared to SLOOP$^-$. We see that ranks saves around 25% time and 30%–40% communication. After some breakdown analysis, we found that the intersection primitive and expansion primitive, although with theoretically linear costs, contributed considerably to the total cost, due to their large hidden constant introduced by number of attributes and LowMC circuits [34]. We believe the improvement factor of SLOOP will be larger once the hidden constants of intersection primitive and expansion primitive are greatly reduced in the future.

We also counted the online costs of computing the output size of these queries in SLOOP: 2s and 90MB, regardless of the value of k, as it is a free-connex query with input size $n = 3 \times 24186$ and output size $m = 1$.

TABLE VII: Costs of graph queries varying parameter k

	Parameter			
k	6	5	4	3
m	21151	94920	234827	887494
	Time (s)			
Plaintext	0.10	0.27	0.45	1.15
SLOOP$^-$	10.40	25.40	55.88	202.80
SLOOP	7.78	19.16	41.77	148.26
SLOOP (w/o ranks)	9.26	20.63	43.26	149.73
	Comm (MB)			
Plaintext	2.30	4.56	8.83	28.74
SLOOP$^-$	531.89	1486.21	3390.13	12319.62
SLOOP	356.73	952.30	2085.10	7369.73
SLOOP (w/o ranks)	452.15	1047.72	2180.52	7465.15

VIII. Conclusion

In this paper, we have presented the first join protocol with linear online cost under the three-party model of MPC. We have then extended it to support any free-connex query and built a system prototype SLOOP. One intriguing question is whether linear cost is also achievable with only two servers, which would further lower the deployment effort of such MPC systems. In fact, most of the primitives SLOOP uses have corresponding two-party protocols with same cost, except `permutation`, for which the best practical two-party protocols have $O(n \log n)$ cost [35], and whether this can be brought down to $O(n)$ is still an open problem in the security literature. Any improvement on these primitives will also bring an improvement to design SLOOP under the two-party model.

References

[1] M. Yannakakis, "Algorithms for acyclic database schemes," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*. VLDB Endowment, 1981, p. 82–94.

[2] M. R. Joglekar, R. Puttagunta, and C. Ré, "Ajar: Aggregations and joins over annotated relations," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2016, p. 91–106.

[3] M. Abo Khamis, H. Q. Ngo, and A. Rudra, "Faq: Questions asked frequently," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2016, p. 13–28.

[4] G. Bagan, A. Durand, and E. Grandjean, "On acyclic conjunctive queries and constant delay enumeration," in *Computer Science Logic*, J. Duparc and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 208–222.

[5] P. Mohassel, P. Rindal, and M. Rosulek, "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, 2020, p. 1271–1287.

[6] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: A maliciously-secure MPC platform for collaborative analytics," in *Proceedings of the 30th Conference on USENIX Security Symposium*, 2021.

[7] G. Asharov, K. Hamada, R. Kikuchi, A. Nof, B. Pinkas, and J. Tomida, "Secure statistical analysis on multiple datasets: Join and group-by," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, p. 3298–3312.

[8] S. Badrinarayanan, S. Das, G. Garimella, S. Raghuraman, and P. Rindal, "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, 2022, p. 209–222.

[9] F. Han, L. Zhang, H. Feng, W. Liu, and X. Li, "Scape: Scalable collaborative analytics system on private database with malicious security," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 1740–1753.

[10] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers, "Smcql: Secure querying for federated databases," *Proceedings of the VLDB Endowment*, vol. 10, no. 6, p. 673–684, 2017.

[11] J. Liagouris, V. Kalavri, M. Faisal, and M. Varia, "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*, M. Balakrishnan and M. Ghobadi, Eds. USENIX Association, 2023, pp. 1031–1056.

[12] Y. Wang and K. Yi, "Query evaluation by circuits," in *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. Association for Computing Machinery, 2022, p. 67–78.

[13] ——, "Secure yannakakis: Join-aggregate queries over private data," in *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, 2021, p. 1969–1981.

[14] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *Advances in Cryptology – EUROCRYPT 2020*, A. Canteaut and Y. Ishai, Eds. Cham: Springer International Publishing, 2020, pp. 44–75.

[15] A. Lazzaretti and C. Papamanthou, "Single pass Client-Preprocessing private information retrieval," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5967–5984. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/lazzaretti

[16] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: Extremely simple, single-server pir with sublinear server computation," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 4296–4314.

[17] A. Beimel, Y. Ishai, and T. Malkin, "Reducing the servers computation in private information retrieval: Pir with preprocessing," in *Advances in Cryptology — CRYPTO 2000*, M. Bellare, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 55–73.

[18] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Advances in Cryptology — CRYPTO '91*, 1992, pp. 420–432.

[19] Q. Luo, Y. Wang, K. Yi, S. Wang, and F. Li, "Secure sampling for approximate multi-party query processing," *Proc. ACM Manag. Data*, vol. 1, no. 3, 2023.

[20] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, "Practically efficient multi-party sorting protocols from comparison sort algorithms," in *Information Security and Cryptology – ICISC 2012*, 2013, pp. 202–216.

[21] G. Asharov, K. Hamada, D. Ikarashi, R. Kikuchi, A. Nof, B. Pinkas, K. Takahashi, and J. Tomida, "Efficient secure three-party sorting with applications to data analysis and heavy hitters," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2022, p. 125–138.

[22] D. Evans, V. Kolesnikov, and M. Rosulek, *A Pragmatic Introduction to Secure Multi-Party Computation*, 2018.

[23] P. Mohassel and P. Rindal, "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, 2018, p. 35–52.

[24] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2016, p. 805–817.

[25] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, p. 831–838, 1980.

[26] T. Araki, J. Furukawa, K. Ohara, B. Pinkas, H. Rosemarin, and H. Tsuchida, "Secure graph analysis at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2021, p. 610–629.

[27] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptol.*, vol. 13, no. 1, p. 143–202, jan 2000.

[28] O. Catrina and A. Saxena, "Secure computation with fixed-point numbers," in *Financial Cryptography and Data Security*, R. Sion, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–50.

[29] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2132–2145, 2020.

[30] D. Olteanu and J. Závodný, "Size bounds for factorised representations of query results," *ACM Trans. Database Syst.*, vol. 40, no. 1, mar 2015.

[31] "https://github.com/ladnir/aby3," ABY3.

[32] S. Kumar, F. Spezzano, V. Subrahmanian, and C. Faloutsos, "Edge weight prediction in weighted signed networks," in *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE, 2016, pp. 221–230.

[33] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian, "Rev2: Fraudulent user prediction in rating platforms," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 2018, pp. 333–341.

[34] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for mpc and fhe," in *Advances in Cryptology – EUROCRYPT 2015*. Springer Berlin Heidelberg, 2015, pp. 430–454.

[35] M. Chase, E. Ghosh, and O. Poburinnaya, "Secret-shared shuffle," in *Advances in Cryptology – ASIACRYPT 2020*, 2020, pp. 342–372.