Oblivious Yannakakis: Join-Aggregate Queries over Private Data

Anonymous Author(s)

ABSTRACT

In this paper, we describe an *oblivious* version of the classical Yannakakis algorithm for computing free-connex join-aggregate queries. This protocol can be used in the *secure two-party computation* model, where the parties would like to evaluate a query without revealing their own data. Our protocol presents a dramatic improvement over the state-of-the-art protocol based on Yao's garbled circuit. In theory, its cost (both running time and communication) is linear in data size and polynomial in query size, whereas that of the garbled circuit is polynomial in data size and exponential in query size. In practice, we reduce the running time from years to minutes, as tested on a number of TPC-H queries of varying complexity.

ACM Reference Format:

Anonymous Author(s). 2018. Oblivious Yannakakis: Join-Aggregate Queries over Private Data. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Privacy concerns have become the main hurdle for data-hungry applications, most notably query processing, data analytics, and machine learning. The blueprint for such applications is a *private data federation* [5, 16], in which multiple autonomous data systems work together to provide query answering services through a common interface. However, the key technical challenge in realizing this blueprint is how to address the privacy concerns of the individual data owners. Before formally defining the problem, let's look at a concrete example.

Example 1.1. Consider the following (oversimplified) scenario where an insurance company wishes to estimate the amount of payment it would pay out, classified by disease types, before the patients submit claims. The company's data is stored in two relations R_1(person, coinsurance, state) and R_3(disease, class). On the other hand, the medical records are stored in the hospital's database as a relation R_2(person, disease, cost). If all three relations were available, one would write the following SQL query:

```
select class, sum(cost * (1 - coinsurance))
from R1, R2, R3
where R1.person=R2.person and R2.disease=R3.disease
group by class;
```

The challenge in evaluating this query is that the three relations are held by two parties respectively as their private data. Privacy-preserving query processing protocols need to be designed so that the insurance company can learn the query results, but nothing else about the hospital's data. Meanwhile, the hospital cannot learn anything about the insurance company's data. □

There are two aspects when it comes to privacy protection when evaluating a query like above. First, privacy cannot be protected completely, as the query results must reveal *some* information about the data, which is actually the purpose of asking the query in the first place. Thus, the literature defines privacy as any information beyond what can be inferred from the query results, and it is required that no such information is derivable from the transcript of the query evaluation protocol. Such a protocol is said to be *oblivious*. This notion can be more formally defined using the real-ideal world paradigm (see Section 4), and have been studied in the area of *secure multi-party computation (SMC)* [13].

What if we also want to guard against the query results from releasing private information? As complete privacy cannot be achieved (unless we do not evaluate any query at all), there must be a privacy-utility trade-off. The idea is to add some noise to the query results so that the amount of private information that can be extracted from the perturbed query results is limited: the more noise that is added, the less private information that can be extracted. There has also been extensive study on this aspect, with the dominating approach in recent years being *differential privacy* [12].

These two aspects aim at preventing different data flows from breaching privacy: SMC guards against the transcript of the protocol, i.e., the process of query evaluation, while differential privacy guards against the end results. They are thus complementary, and one could employ both strategies to achieve full-range privacy protection, as long as the noise-addition process in differential privacy can also be made oblivious.

This paper studies the first aspect. In particular, we consider the two-party setting (also known as the 2PC model). Per tradition, we name the two parties as Alice and Bob. We assume that the global schema of the database is publicly known, but each relation is held by either Alice or Bob as her/his private data. We aim at designing oblivious protocols for evaluating free-connex join-aggregate queries (see Section 3 for a more formal definition). Joins are important for obvious reasons; aggregations are also necessary in the SMC model, as SMC only protects privacy beyond the query results. If the query results were too numerous, they unavoidably would reveal a lot of private information. Furthermore, if one also wants to use differential privacy to guard against the query results, the results can only be aggregates since all noise-addition techniques work only for aggregates. In fact, returning an original tuple in the query results cannot possibly be differentially private.

1.1 Previous Work

The SMC model was first proposed by Yao [32]. Over the years, it has gradually developed from a theoretical curiosity to a practical

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

tool for privacy-preserving applications. In the 2PC model, the most popular approach is Yao's *garbled circuit* [32], which is a generic protocol that can be used to evaluate any function obliviously by expressing it as a Boolean circuit. Its cost (both computation and communication) is proportional to the size of the circuit, and the constant coefficient has been reduced significantly over the years, thanks to improvement in both hardware and algorithm design [13].

Bater et al. [5] have developed *SMCQL*, the first two-party oblivious query processing engine that supports joins, following the *ObliVM* [21] framework. It processes a given SQL query in a bottom-up fashion following an optimized query plan, evaluating each relational operator using a garbled circuit generated by ObliVM. However, since the SMC model forbids the transcript of the protocol to reveal any information about the private data beyond what can be learned from the query results, the intermediate result size of each relational operator must be hid from the parties. Thus, each intermediate relation has to be padded with dummy tuples to reach its maximum possible size, so that it doesn't depend on the actual input data. For a join over k relations each of size N, the maximum possible intermediate result size is $O(N^k)$, even though the final query result may just consist of a few aggregates. Therefore, the garbled circuit has size $\tilde{O}(N^k)$[1]. Evaluating such a huge garbled circuit is very expensive. In practice, it takes a day to evaluate a query on two relations with only hundreds of tuples, as shown in their experiments.

To mitigate the problem, Bater et al. [6] in a follow-up work proposed a method to reduce the intermediate result size. Instead of using the maximum possible intermediate result size, they use the actual size, which is much smaller (except on contrived inputs). However, as the actual intermediate result size could reveal private information, they pad a random number of dummy tuples, where the number of dummy tuples is determined by differential privacy. This has drastically reduced the cost, but it deviates from the SMC model, which requires that no information be leaked other than the end query results. Note that the intermediate result size (even perturbed with noise) contains information *not* in the end query results. Consider the query in Example 1.1, and suppose the query plan joins R_1 and R_2 first. The intermediate join size $|R_1 \bowtie R_2|$, even with noise, can give the insurance company an estimate on the average number of diseases a person have treated at the hospital, something the hospital may not be willing to reveal. Therefore, this approach violates the requirement of SMC, although it still respects differential privacy.

A more direct approach is to design non-circuit-based protocols. However, success so far has been limited to some specialized problems. In particular, the *private set intersection (PSI)* problem [9, 24–26] has received a lot of attention. Here, Alice and Bob each hold a set and they would like to find the common elements of the two sets, but those belonging only to one set should not be revealed to the other party. The specialized PSI protocols have cost linear to the total size of the two sets. This is a significant reduction in cost, compared with a circuit that requires quadratic size for computing set intersection. Note that set intersection is exactly a degenerated join $R_1(A) \bowtie R_2(A)$, so our protocol can be considered as solving a (much) more general problem than PSI, still with linear cost.

[1]The $\tilde{O}$ notation hides polynomial dependencies on $\log N$, the query size (i.e., number of relations and attributes), the security parameters κ, σ, and the bit-length of attributes ℓ. Please see Section 3 and 4 for formal definitions of these parameters.

1.2 Our Results

We present *oblivious Yannakakis*, a 2PC protocol for evaluating any *free-connex join-aggregate* query (formal definition given in Section 3) with $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$ cost (both computation and communication), where IN is the total number of tuples in the input relations and OUT is the output size. Note that OUT refers to the size of the final query output after aggregation, not the join size. Thus OUT is usually much smaller than IN. However, it is also possible that OUT > IN (e.g., when the query has many attributes in the `group by` clause), although this may not be common in applications with privacy concerns. But in either case, an $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$ cost is clearly optimal, even in the non-private setting.

The oblivious Yannakakis protocol works in the strict 2PC model. It reveals nothing other than what can be inferred from the query results. It does not assume any trusted third party, and all data is kept secret to their respective owners. Thus, it represents a significant improvement over the generic, circuit-based approach taken by SMCQL. In theory, its data complexity (i.e., dependency on IN) has been reduced from a large polynomial $\tilde{O}(N^k) = \tilde{O}\left(\left(\frac{\mathrm{IN}}{k}\right)^k\right)$ to linear, and the combined complexity (i.e., dependency on both IN and k) reduced to polynomial from exponential. This translates to a mind-boggling improvement in practice as demonstrated by our experimental results in Section 8.

Our starting point is the observation that an oblivious protocol requires that its transcript to be indistinguishable when running on different inputs. This implies that the cost of the protocol should not depend on the input, which effectively means that every input should incur the same cost as that on the worst-case input. This renders all the cost-based query optimization techniques useless. Instead, we should look for worst-case efficient algorithms, whose costs are bounded no matter what the input is. The Yannakakis algorithm [31], a classical algorithm for evaluating acyclic queries with worst-case running time $O(\mathrm{IN} + \mathrm{OUT})$, is exactly such an algorithm.

However, porting the Yannakakis algorithm to the SMC model is nontrivial, which is why it hasn't been done 40 years after its invention. The first technical difficulty is that it is not a circuit-based algorithm; it heavily relies on hash joins to achieve the optimal $O(\mathrm{IN} + \mathrm{OUT})$ running time. To overcome this difficulty, we design non-circuit-based protocols for joins and semijoins, whose cost is linear to the input size plus output size. Another hurdle, which is more subtle, is that we have to put the protocols for different relational operators together, in a way such that the intermediate results do not reveal any private information. This is challenging in our setting, since while our join and semijoin protocols are non-circuit-based, our aggregation protocol still is. To solve this problem, we make use of a very recent PSI protocol [25] that is "circuit-friendly" in designing our semijoin protocols. Finally, we use *secret sharing* and *oblivious extended permutation (OEP)* as "glue" to assemble the pieces together, yielding the oblivious Yannakakis protocol. Note that the assembly process is much easier in SMCQL, where all operators use garbled circuits.

The rest of the paper is organized as follows. After surveying other related work, we precisely define the class of queries that can be handled by our protocol in Section 3, which also includes a brief review of the classical Yannakakis algorithm. In fact, we have to make some modifications to the original algorithm in order to port it to the SMC model. Section 4 formally defines the security model and the oblivious query evaluation problem. Section 5 reviews some cryptographic primitives, as well as their adaptions to fit our purpose. The oblivious Yannakakis algorithm is presented in Section 6, with some of its extensions discussed in Section 7. We present our experimental study in Section 8 before introducing some future work.

2 RELATED WORK

Efficient and secure protocols for many key operations on secret-shared databases are introduced in [20], but they did not study joins, the most important operation. Laur et al. [19] implemented an oblivious AES protocol based on Sharemind, and used it to securely compute 2-way joins. However, their join protocol cannot be combined with other operators (including other joins), as it reveals the intermediate join size. Besides, it requires degree constraints on the data. When no such constraints exist, its cost is quadratic to the input size, which is the same as that of the garbled circuit. On the other hand, our protocol assumes no constraints at all on the data.

Another approach to reducing the high cost of garbled circuits is to assume a trusted third party [1, 29, 30]. Note that if this third party could be trusted with all data, the problem would not exist as this party can simply evaluate the query and send the results back. So the model allows the trusted party to access a subset of the columns. When the trusted party have access to certain columns, especially the join attributes, this approach significantly improves query efficiency. However, there is no improvement when all columns must be kept secret. Hardware vendors now offer chips, such as Intel SGX, that can be considered as such a trusted party. They are being increasingly adopted [2, 4, 28] due to great reductions in execution costs. Our protocol is entirely software-based and assumes no trusted entities at all, but it's possible to shift some of the computation to a trusted party, if one exists, to further reduce the cost.

Finally, another popular model studied in the literature is *outsourced databases*, where the data owner uploads encrypted data to a cloud, who provides SQL services to the data owner. Representative systems include CryptDB [27] and Cipherbase [2]. This is distinct from the SMC model, where multiple data owners would like to query on their joint data without sharing them.

3 JOIN-AGGREGATE QUERIES OVER ANNOTATED RELATIONS

3.1 Query Definition

Hypergraph and Join. A (natural) join can be modeled as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where the vertices $\mathcal{V}$ corresponds to the attributes and each hyperedge $F \in \mathcal{E}$ corresponds to a relation. Let $\mathcal{D}^A$ be the domain of attribute $A \in \mathcal{V}$, from which its values are drawn. For each hyperedge $F \in \mathcal{E}$, there is a relation R_F that consists of a set of tuples, where each tuple assigns a value from $\mathcal{D}^A$ to A for each attribute $A \in F$. We also use the notation such as $R(A, B, C)$ to indicate that the attributes of R are A, B, C.

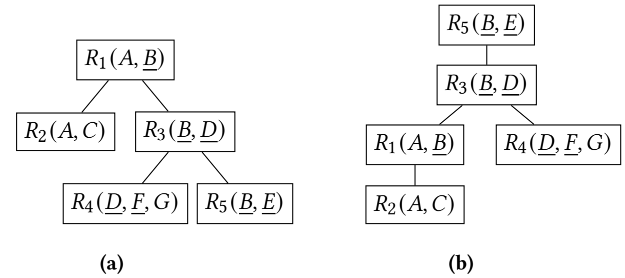

Figure 1: An acyclic join with different join trees, with the output attributes underlined. The one in (b) testifies that it's free-connex.

The join results $\mathcal{J} = \bowtie_{F \in \mathcal{E}} R_F$ are all tuples that are consistent with some tuple in R_F for every $F \in \mathcal{E}$, i.e.,

$$\mathcal{J} = \{t \in \mathcal{D}^{\mathcal{V}} \mid \forall F \in \mathcal{E} : \pi_F(t) \in R_F\}.$$

Join Tree. A join tree of a hypergraph $\mathcal{H}$ is a tree $\mathcal{T}$ where the hyperedges of $\mathcal{H}$ are the nodes of $\mathcal{T}$, such that for each attribute $A \in \mathcal{V}$, all nodes containing A are connected in $\mathcal{T}$. The join is said to be *acyclic* if its hypergraph has a join tree. For example, the query in Example 1.1 is acyclic, with a join tree $R_1 - R_2 - R_3$. Figure 1 shows a more complicated example. On the other hand, the triangle join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, C)$ is not acyclic.

Since there is a one-to-one correspondence, we will not distinguish between a node in $\mathcal{T}$ and the relation it corresponds to.

Annotated Relations. We follow the terminology from [17]. Let $(S, \oplus, \otimes)$ be a finite commutative semiring, where S is the ground set and $\oplus$ and $\otimes$ are its "addition" and "multiplication" operators. In our paper, we simply take the ground set S to be $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$, $n = 2^\ell$, where ℓ is the least number of bits to represent all annotations. This is without loss of generality as they are merely identifiers of the semiring elements. The only requirements we impose are (1) 0 is the $\oplus$-identity of the semiring; (2) 1 is the $\otimes$-identity of the semiring; and (3) $\oplus$ and $\otimes$ can be evaluated by $\tilde{O}(1)$-size Boolean circuits. For example, the semiring $(\{\mathsf{True}, \mathsf{False}\}, \vee, \wedge)$ can be trivially handled by constant-size circuits by mapping True to 1 and False to 0; for the semiring (actually, ring) $(\mathbb{Z}_n, +, \times)$, where operations are done modulo n, the circuit for $+$ has size $O(\log n)$ while $O(\log^2 n)$ for $\times$.

Given a semiring, we associate each tuple t with an annotation $v(t) \in S$, and extend the join and projection-aggregation operations to annotated relations as follows. The *annotated join* $\bowtie^{\otimes}_{F \in \mathcal{E}} R_F$, in addition to computing the join results $\mathcal{J}$, also computes the $\otimes$-aggregate of the annotations of the tuples compromising each join result t as its annotation, i.e.,

$$v(t) = \bigotimes_{F \in \mathcal{E}} v(\pi_F(t)).$$

An *annotated projection-aggregation* $\pi^{\oplus}_F(R)$ first performs a normal projection, i.e., finds all the distinct (combinations of) values on F in R. Then for each distinct value, it computes the $\oplus$-aggregate of annotations of all tuples in R with that distinct value. More precisely, for any $t \in \pi_F(R)$, its annotation in $\pi_F^{\oplus}(R)$ is

$$v(t) = \bigoplus_{r \in R: \pi_F(r) = t} v(r).$$

Note that $\pi_F^{\oplus}(R)$ exactly corresponds to

select $\oplus$ (annotation) group by F

in SQL. In particular, when $F = \emptyset$, $\pi_F^{\oplus}(R)$ returns a single empty tuple, whose annotation is the $\oplus$-aggregate of all annotations of tuples in R.

Define $\pi_F^1(R) := \pi_F(\{t \in R \mid v(t) \neq 0\})$, while the annotations of all tuples in $\pi_F^1(R)$ are set to 1. Then we define an annotated semijoin as

$$R_F \ltimes^{\otimes} R_{F'} := R_F \bowtie^{\otimes} \pi^1_{F \cap F'}(R_{F'}),$$

namely, it returns the subset of tuples in R_F, which would produce at least one nonzero-annotated join result if joined with $R_{F'}$. However, the semijoin itself does not actually do the join; instead, it simply finds this subset while preserving their annotations in R_F.

Join-Aggregate Queries. Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a set of output attributes $O \subseteq \mathcal{V}$, and a set of annotated relations $R_F, F \in \mathcal{E}$, a *join-aggregate* query is $Q = \pi_O^{\oplus}(\bowtie^{\otimes}_{F \in \mathcal{E}} R_F)$. We use IN $= \sum_{F \in \mathcal{E}} |R_F|$ to denote the total size of all input relations, and OUT the output size; note that both IN and OUT can be much smaller than the join size $|\mathcal{J}|$.

By appropriately defining the semiring and the annotations, one can express many SQL queries as join-aggregate queries.

Example 3.1. To answer the query in Example 1.1, we can use the semiring $(\mathbb{Z}_n, +, \times)$, where n is chosen large enough so that there will be no overflows. The annotation of each tuple in R_1 is set to `100*(1-coinsurance)`, assuming `coinsurance` is a floating-point number with 2 digits of precision. Tuples in R_2 have annotations equal to `cost`, and the annotations of all tuples in R_3 are set to 1. The output attribute is $O = \{\texttt{class}\}$. Then the query becomes a join-aggregate query, except that we need to scale the query results down by 100. □

We consider *free-connex* join-aggregate queries [3], namely (1) the hypergraph $\mathcal{H}$ is acyclic; and (2) $\mathcal{H}$ has a join tree $\mathcal{T}$ with a designated root node, such that for any $A \in O$ and $B \in \mathcal{V} - O$, TOP(B) is not an ancestor of TOP(A) in $\mathcal{T}$, where TOP(X) denotes the highest node in $\mathcal{T}$ containing attribute X. For example, the query in Example 1.1 is free-connex, using the join tree $R_3 - R_2 - R_1$ with R_3 as the root. On the other hand, if we do group-by on $\{\texttt{class}, \texttt{coinsurance}\}$, then the query will not be free-connex. Note that if $O = \emptyset$, condition (2) is automatically satisfied. As another example, consider the query in Figure 1a when $O = \{B, D, E, F\}$. This join tree does not satisfy condition (2), but the one in Figure 1b does. So the query is still free-connex.

In fact, free-connex queries are exactly the class of join-aggregate queries that are known to be solvable in $\tilde{O}(\text{IN} + \text{OUT})$ time [17], even in the centralized, non-private setting. The problem is wide open for non-free-connex queries. In fact, the simplest non-free-connex query, $R_1(A, B) \bowtie R_2(B, C)$ with $O = \{A, C\}$, can already express matrix multiplication, and people are still looking for a linear-time algorithm after 50 years of research.

3.2 Yannakakis Algorithm

The Yannakakis algorithm [31] is a classical algorithm that evaluates an acyclic join query in $O(\text{IN} + \text{OUT})$ time. It has been extended to handle free-connex join-aggregate queries by Joglekar et al. [17]. In order to use it for oblivious query evaluation, we modify it into the following 3-phase algorithm. Given an free-connex join-aggregate query Q, let $\mathcal{T}$ be its join tree satisfying condition (2) stated above.

(1) *Reduce.* We first reduce the query by removing all its non-output attributes. This is done in a bottom-up pass on $\mathcal{T}$ while performing joins and aggregations. For each node R_F with parent F_p, let $F' = (O \cup F_p) \cap F$. If $F' \subseteq F_p$, we update R_{F_p} as $R_{F_p} \leftarrow R_{F_p} \bowtie^{\otimes} \pi^{\oplus}_{F'}(R_F)$ and then remove the node R_F from $\mathcal{T}$. If $F' - F_p \neq \emptyset$, the reduce process stops going upward. Instead, we only update R_F as $R_F \leftarrow \pi^{\oplus}_{F'}(R_F)$. Note that the attributes of R_F are also updated to $F \leftarrow F'$. In this case, all attributes in F' must be output attributes, so are all the ancestors of R_F due to the free-connex property. Therefore, after the reduce phase, only output attributes remain.
(2) *Semijoin.* We use two passes of semijoins to remove the *dangling tuples*, i.e., tuples that do not appear in the join results. In the first pass, we visit the nodes in the reduced join tree $\mathcal{T}$ in some bottom-up order. For any non-root node R_F with parent R_{F_p}, we update R_{F_p} as $R_{F_p} \leftarrow R_{F_p} \ltimes^{\otimes} R_F$. Note that the semijoin just returns a subset of the tuples in R_{F_p}, but does not change their annotations. Then we perform a top-down pass in a similar fashion, updating each node R_F as $R_F \leftarrow R_F \ltimes^{\otimes} R_{F_p}$.
(3) *Full join.* Finally, we compute the join and their annotations in a bottom-up pass, i.e., for each non-root node R_F with parent R_{F_p}, we update R_{F_p} as $R_{F_p} \leftarrow R_{F_p} \bowtie^{\otimes} R_F$ and remove R_F. The root relation is the query results after this phase terminates.

Example 3.2. Below we illustrate how the Yannakakis algorithm works on the query shown in Figure 1b.

(1) *Reduce.* First, consider $R_2(A, C)$ and its parent $R_1(A, B)$. We reduce the node R_2 after updating $R_1 \leftarrow R_1 \bowtie^{\otimes} \pi^{\oplus}_A(R_2)$. Similarly, we then reduce the node R_1 after updating $R_3 \leftarrow R_3 \bowtie^{\otimes} \pi^{\oplus}_B(R_1)$. Afterwards, we remove the non-output attribute G in R_4 by updating $R_4 \leftarrow \pi^{\oplus}_{D,F}(R_4)$. Now, three nodes R_4, R_3 and R_5 remain, which only contain output attributes.
(2) *Semijoin.* We perform a bottom-up pass: $R_3 \leftarrow R_3 \ltimes^{\otimes} R_4$, $R_5 \leftarrow R_5 \ltimes^{\otimes} R_3$, followed by a top-down pass: $R_3 \leftarrow R_3 \ltimes^{\otimes} R_5$, $R_4 \leftarrow R_4 \ltimes^{\otimes} R_3$, so that dangling tuples are removed.
(3) *Full join.* Finally, we compute $R_4 \bowtie^{\otimes} R_3 \bowtie^{\otimes} R_5$ as the results of the query. □

Correctness. The original Yannakakis algorithm has two phases: the semijoin phase and the join-aggregate phase. Our modified version splits the join-aggregate phase into the reduce phase and the full join phase, and pulls the reduce phase in front of the semijoin phase (for reasons that will become clear later). This modification does not affect the correctness proof of the algorithm [17], which shows that the query results are preserved after each individual join, semijoin, and aggregation operation.

Complexity analysis. The complexity analysis of the modified algorithm is also similar as the original algorithm. First, by building appropriate indexes, a semijoin, a join, and a projection (including their annotated versions) can all be done in time proportional to its input and output size. Thus, it is sufficient to bound all the intermediate result sizes during the three phases of computation. Consider the reduce phase. The aggregation $\pi^{\oplus}_{F'}(R_F)$ obviously does not enlarge the size of R_F. The join $R_{F_p} \leftarrow R_{F_p} \bowtie^{\otimes} \pi^{\oplus}_{F'}(R_F)$ does not make R_{F_p} larger, either, since $F' \subseteq F_p$ (so it is basically a semijoin). So this phase can be done in time $O(\text{IN})$. The semijoin phase obviously cannot produce more tuples, so it can be done in time $O(\text{IN})$, too. For the full join phase, since all remaining attributes are output attributes and dangling tuples have been removed, every intermediate result must be part of a final output tuple. Thus, any intermediate result size is bounded by $O(\text{OUT})$.

4 OBLIVIOUS QUERY EVALUATION

In this section, we formalize the problem of oblivious query processing in the security two-party computation (2PC) model. Alice and Bob each hold some relations as their private data, and agree to compute a free-connex join-aggregate query jointly. The agreement also includes a designated receiver (Alice or Bob, or both) who will get the query results. Without loss of generality, we assume Alice is the only receiver; if Bob is also a receiver, Alice can simply forward the query results to Bob at the end of the protocol. We assume that the database schema, the query, the input relation sizes, as well as the output size are public knowledge. If the relation sizes are sensitive, one could pad *dummy tuples*[2] and release the relation size after the padding. In this case, however, the input size IN must also include these dummy tuples. Similarly, we can add dummy output tuples if the true output size is sensitive.

Adversary. The security of a protocol must be measured against some adversary, and various adversary models have been studied in the literature. Broadly speaking, there are two types of adversaries: *semi-honest* and *malicious*. In the semi-honest (a.k.a. honest but curious) model, Alice and Bob will follow the prescribed protocol, but will try to learn information about the other party's data from the transcript of the protocol. Thus, protocols designed in this model are often said to be *oblivious*, i.e., their transcripts do not reveal any information other than the query results. This model is suitable for scenarios where both parties can guarantee their correct execution of the protocol, but there are data leakage threats, e.g., internal employees spying on the protocol or hackers stealing information.

In addition to observing the transcript, a malicious party may deviate from the protocol arbitrarily, in an attempt to learn the other party's private data. This is a much stronger security model, and it often requires more costly protocols. Another point to note is that, since a malicious party can deviate from the protocol arbitrarily, he may completely change his own data. Thus, the query results are not guaranteed to be correct in this model, unlike in the semi-honest model. We will only study the semi-honest model in this paper. Nevertheless, there are generic approaches that can be used to *harden* a semi-honest protocol into a malicious-secure protocol, such as cut-and-choose [8], zero-knowledge proofs [14], BDOZ [7] and SPDZ [10]. These hardening techniques in principle should also apply to our protocol, but we leave the details to future work.

[2]We can reserve a special region in the domain of each attribute to draw dummy tuples from. All dummy tuples have annotation 0, and have different attribute values, which are also distinct from true tuples, so that they do not join with any other tuple.

Security Definition. To formalize the notion of "not learning any information beyond the query results", we introduce the real-ideal world paradigm. In the *ideal world*, Alice and Bob send their data to a trusted third party, who evaluates the query, and returns the results to Alice (the designated receiver); Bob gets no output. In the *real world*, they follow the prescribed protocol to evaluate the query. The *view* of a party (in either the ideal and real world) consists of all messages s/he has sent and received during the protocol, plus her/his own input and output. The protocol is secure if, for any input and any adversary $\mathcal{A}$ in the real world, there exists a simulator such that, given $\mathcal{A}$'s ideal-world view, the simulator can produce a view that is indistinguishable from $\mathcal{A}$'s real-world view. This means that everything $\mathcal{A}$ sees in the real world can be completely created from what s/he sees in the ideal world, which simply consists of her/his own input and output.

Security Parameters. We still need to formalize the *indistinguishability* between two views, which can be perfect, statistical, or computational. Note that oblivious protocols are often randomized, so the simulator must also produce a random view. We have perfect indistinguishability if the distributions of the real-world view and the simulated view are identical; otherwise, we say they are statistically indistinguishable if the statistical distance (e.g. total variation) between the two distributions is smaller than $2^{-\sigma}$, where σ is the *statistical security parameter*. Statistical indistinguishability does not restrict the computing power of the adversary (thus it is also called *unconditional security*).

For computational indistinguishability, we feed the view to a probabilistic polynomial-time distinguisher, who tries to decide if it is a real-world view or a simulated view, and we require that the success probability be no more than $1/2+2^{-\sigma}$. Computational indistinguishability is often based on hardness assumptions on certain problems like pseudo-random functions, discrete logarithm, and factoring. The *computational security parameters* κ refers to the key length used in cryptographic primitives to achieve computational indistinguishability.

Lastly, a protocol is also allowed to fail, i.e., it terminates without computing the correct output. This probability is also set to $2^{-\sigma}$, the same as that of breaching security.

All cryptographic primitives used in our protocol have costs polynomial in ℓ, σ, κ, and we hide their dependency in the $\tilde{O}$ notation to simplify the expression. In practice, $\sigma = 40$, $\kappa = 128$ (for symmetric encryption) or 1024 (for asymmetric encryption) are considered sufficient in most applications.

5 CRYPTOGRAPHIC PRIMITIVES

5.1 Secret Sharing

A secret sharing scheme partitions a secret value $v \in \mathbb{Z}_n$ into two *shares*, such that they can be used to reconstruct v, but neither alone reveals any information about v. We use the notation $[\![v]\!]$ to stress that v has been shared, and use $[\![v]\!]_1$ and $[\![v]\!]_2$ to represent the share owned by Alice and Bob respectively. In this paper, we use the following simple scheme, known as *arithmetic* *sharing*: pick $[\![v]\!]_1$ uniformly at random from $\mathbb{Z}_n$ and set $[\![v]\!]_2 = (v - [\![v]\!]_1) \bmod n$. It is clear that the two shares can reconstruct the secret as $([\![v]\!]_1 + [\![v]\!]_2) \bmod n = v$. Meanwhile, $[\![v]\!]_1$ and $[\![v]\!]_2$ are both uniformly random numbers, so they reveal nothing about v. In the sequel, all the arithmetic operations are done modulo n, unless stated otherwise.

Our oblivious Yannakakis algorithm is composed of oblivious protocols for individual relational operators. However, a key difficulty is that we are not allowed to reveal the intermediate results (including their sizes and access patterns). Thus, we will hide all the intermediate results using secret sharing, which means that the individual relational operators may need to take inputs that are secret-shared, and also produce outputs in shared form. In this case, we simply say that the input to the operator is $[\![v]\!]$ and the output is $[\![u]\!]$, meaning that the Alice inputs $[\![v]\!]_1$, Bob inputs $[\![v]\!]_2$, while they obtain $[\![u]\!]_1$ and $[\![u]\!]_2$ as their respective output. The oblivious protocol for the operator will make sure that neither learns the other's share when evaluating the operator. As a simple example, computing $z = x + y$ can be done easily in the shared form (actually without any communication): Alice simply computes $[\![z]\!]_1 = [\![x]\!]_1 + [\![y]\!]_1$, and Bob computes $[\![z]\!]_2 = [\![x]\!]_2 + [\![y]\!]_2$. It can be verified that $[\![z]\!]_1 + [\![z]\!]_2 = [\![x]\!]_1 + [\![x]\!]_2 + [\![y]\!]_1 + [\![y]\!]_2 = x + y$, i.e., $[\![z]\!]_1$ and $[\![z]\!]_2$ form a valid secret sharing of z.

A value v can be converted to $[\![v]\!]$ easily: Suppose Alice holds a value v. She just picks a random $[\![v]\!]_1$ and sends $[\![v]\!]_2 = v - [\![v]\!]_1$ to Bob. Conversely, to go from $[\![v]\!]$ to v, we ask one party, say Bob, to send his share to Alice. We call this operation *revealing* v to Alice. In our protocol, we will only reveal non-private information, which includes the query results (or anything that can be inferred from the query results), random numbers, or ciphertext.

5.2 Garbled Circuits

As mentioned, garbled circuits [32] provide a generic 2PC solution. While it is inefficient to express the whole query as a gigantic circuit, we still make use of small garbled circuits for key operations in our protocol. We will not elaborate on how it works (see [13] for an excellent description), but only define its input and output. Given a function expressed as a Boolean circuit (the circuit is public knowledge) and Alice and Bob's private data (which may be the shares of secret values), the garbled circuit protocol obliviously evaluates function, and obtains the output in secret-shared form. A garbled circuit can be evaluated with communication cost and running time both $\tilde{O}(\text{size of circuit})$, and it requires a constant number of communication rounds.

One technicality is that garbled circuits are Boolean circuits, while the input and output used in our protocol are integers drawn from $\mathbb{Z}_n$. While input integers can be converted to $\log n$ Boolean values straightforwardly, the output Boolean values of the garbled circuit are shared using *Yao's secret sharing*. Fortunately, there is a simple technique [11] that can convert an integer whose bits are Yao-shared to an arithmetically shared form as described above.

5.3 Private Set Intersection (PSI)

In the *private set intersection (PSI)* problem, Alice has a set X with size M and Bob has a set Y with size N, and the goal is to compute $X \cap Y$. Most PSI protocols in the literature reveal the output $X \cap Y$ to Alice, which is fine and actually required by the PSI problem. However, we will be using PSI to produce intermediate results that cannot be revealed; instead, the output should only be obtained in its secret-shared form, which will be further processed by other operators.

The recent PSI protocol of Pinkas et al. [25] fits our purpose, which runs in a constant number of rounds and has $\tilde{O}(M + N)$ running time and communication. First, Alice picks 3 random independent hash functions to build a *cuckoo hash table* [23] with $B = O(M)$ bins[3] on X. The details of cuckoo hashing are not important for understanding how to use this PSI protocol. All we need to know is that each element in X is mapped to one of the 3 locations specified by the 3 hash functions, and with probability at least $1 - 2^{\sigma}$, each bin contains at most one element in X. Let x_i be the element in X that is mapped to the i-th bin; x_i is set to a dummy value if the i-th bin is empty. Alice sends the 3 hash functions to Bob, who also builds a hash table on his set Y. Bob does not use cuckoo hashing, but hashes each element in Y to all 3 bins specified by the hash functions. Then they run the PSI protocol on the bins. At the end of the protocol, Alice and Bob obtain $[\![\text{Ind}(x_i \in Y)]\!]$ for each $i \in [B]$, where $\text{Ind}(\cdot)$ is the indicator function.

In addition, the PSI protocol in [25] supports *payload sharing*, which will also be useful. More precisely, Bob has a payload $z_j \in \mathbb{Z}_n$ for each $y_j \in Y$. At the end of the protocol, in addition to $[\![\text{Ind}(x_i \in Y)]\!]$ for each $i \in [B]$, the protocol also returns $[\![z_j]\!]$ if $x_i = y_j$ for some j; otherwise, it returns $[\![0]\!]$[4].

Example 5.1. By using garbled circuits and the PSI protocol, we can already evaluate some simple join-aggregate queries, such as $\pi^{\oplus}_{\texttt{person}}(R_1(\texttt{person}, \texttt{coinsurance}) \bowtie^{\otimes} R_2(\texttt{person}, \texttt{disease}, \texttt{cost}))$, assuming each person has at most one record in R_1 (likely the case) as well as in R_2 (unlikely the case). Let Alice be the insurance company, who is also the designated receiver of the query results, and Bob the hospital. We first run PSI on R_1 and R_2, treating the `person` attributes as elements of the two sets, and the annotations (i.e., the `cost` attributes) as R_2's payloads. Let $t_i^{(1)} \in R_1$ be the tuple in the i-th bin in Alice's cuckoo hash table. The PSI protocol will return, for each $i \in [B]$, $[\![\text{Ind}(t_i^{(1)} \in R_2)]\!]$ and $[\![v(t_j^{(2)})]\!]$, where $v(t_j^{(2)})$ is the annotation of the tuple $t_j^{(2)} \in R_2$ that joins with $t_i^{(1)}$, if such a $t_j^{(2)}$ exists, and 0 otherwise. Next, we build a garbled circuit for each i, where Alice inputs $v(t_i^{(1)})$ (i.e., `100*(1-coinsurance)` of the tuple $t_i^{(1)}$) and $[\![v(t_j^{(2)})]\!]_1$, and Bob inputs $[\![v(t_j^{(2)})]\!]_2$. The circuit computes (the secret shares of)

$$v(t_i^{(1)}) \otimes (([\![v(t_i^{(2)})]\!]_1 + [\![v(t_i^{(2)})]\!]_2)) = v(t_i^{(1)}) \otimes v(t_i^{(2)}),$$

which is the payment the insurance company needs to make for person $t_i^{(1)}$. Finally, we reveal the results of the B garbled circuits to Alice.

Note that even if $t_i^{(1)}$ is a dummy tuple with annotation 0 (and Alice knows it), they still have to evaluate the circuit; otherwise, Bob would know that the i-th bin of the cuckoo hash table is empty,

[3]In practice, having $B = 1.27M$ bins is sufficient.

[4]The original PSI protocol [25] does not directly return $[\![\text{Ind}(x_i \in Y)]\!]$ or $[\![0]\!]$, but they can be obtained by using a garbled circuit on their output.

which could leak information about R_1 to Bob (this can be considered as the access pattern of the output being leaked). For a dummy $t_i^{(1)}$, the garbled circuit will return $[\![0]\!]$, but all Bob receives is $[\![0]\!]_2$, which is just a random number, indistinguishable from $[\![v]\!]_2$ for any real query result v.

It is not surprising that garbled circuits and PSI are enough for this simple query, as under the strong assumption that the `person` attribute is unique in both relations, it is really just set intersection. To handle more general free-connex join-aggregate queries, we need to make relational operators such as semijoin, join, and projection-aggregation oblivious, which we introduce in the next section. □

5.4 Oblivious Extended Permutation (OEP)

Suppose Alice holds a function $\xi : [N] \to [M]$, and Bob holds a length-M sequence $\{x_i\}_{i=1}^M$ where each $x_i \in \mathbb{Z}_n$. The function ξ is also called an *extended permutation*. In the *oblivious extended permutation (OEP)* problem [22], they wish to obliviously map the sequence $\{x_i\}_{i=1}^M$ to a length-N sequence $\{y_i\}_{i=1}^N$ as specified by ξ, i.e., $y_i = x_{\xi(i)}$. The output $\{y_i\}$ must be obtained in a shared form. The OEP protocol of Mohassel and Sadeghian [22] solves the problem with $\tilde{O}(M+N)$ running time and communication cost.

If the sequence $\{[\![x_i]\!]\}_{i=1}^M$ is given in secret-shared form, we can still use OEP to permute it, as follows. Suppose Alice holds the private permutation function $\xi : [N] \to [M]$, and they wish to permute $\{[\![x_i]\!]\}$ while keeping ξ and $\{x_i\}$ private. We invoke OEP on Bob's shares $\{[\![x_i]\!]_2\}$ with ξ, which results in Alice obtaining $[\![[\![x_{\xi(i)}]\!]_2]\!]_1$ and Bob $[\![[\![x_{\xi(i)}]\!]_2]\!]_2$. Alice then locally computes $[\![[\![x_{\xi(i)}]\!]_2]\!]_1 + [\![x_{\xi(i)}]\!]_1$, which along with $[\![[\![x_{\xi(i)}]\!]_2]\!]_2$ forms the shares of $[\![x_{\xi(i)}]\!]$ as required:

$$[\![[\![x_{\xi(i)}]\!]_2]\!]_1 + [\![x_{\xi(i)}]\!]_1 + [\![[\![x_{\xi(i)}]\!]_2]\!]_2 = [\![x_{\xi(i)}]\!]_1 + [\![x_{\xi(i)}]\!]_2 = x_{\xi(i)}.$$

Note that the new shares of $\{x_{\xi(i)}\}$ are generated by the OEP protocol using fresh randomness, so they reveal nothing about the original shares of $\{x_i\}$.

5.5 PSI with Secret-shared Payloads

In Example 5.1, we used PSI to share payloads $\{z_j\}$ associated with Bob's set $Y = \{y_j\}$. In queries involving more than one join, the payloads of intermediate results are not explicitly given, but are secret-shared, i.e., Alice holds $\{[\![z_j]\!]_1\}$ and Bob holds $\{[\![z_j]\!]_2\}$. The trivial way of revealing $\{z_j\}$ to Bob and then running PSI would not work as $\{z_j\}$ can be intermediate results that have been derived from not only Bob's data, thus must be protected. Below we present a protocol with $\tilde{O}(M+N)$ running time and communication to solve the problem.

Let B be the size of Alice's cuckoo hash table in the PSI protocol. First they locally extend the shares $\{[\![z_j]\!]\}_{j=1}^N$ to $\{[\![z_j]\!]\}_{j=1}^{N+B}$ with $[\![z_j]\!]_1 = [\![z_j]\!]_2 = 0$ for $j > N$. Then they use OEP to permute the shares from $\{[\![z_j]\!]\}_{j=1}^{N+B}$ to $\{[\![z'_j]\!]\}_{j=1}^{N+B}$, where $z'_j = z_{\xi_1(j)}$ and $\xi_1 : [N+B] \to [N+B]$ is a random permutation (bijection) locally generated by Bob. Afterwards, they run the PSI protocol of [25] on X and Y, while the payload of y_j is $\xi_1^{-1}(j)$, where ξ_1^{-1} is the inverse permutation of ξ_1. Note that $z'_{\xi_1^{-1}(j)} = z_j$.

Let x_i be the element in the i-th bin of Alice's cuckoo hash table. The PSI protocol will return $[\![\mathrm{Ind}(x_i \in Y)]\!]$ and $[\![\xi_1^{-1}(j)]\!]$, if $x_i = y_j$ for some $y_j \in Y$. We build a garbled circuit with inputs $[\![\mathrm{Ind}(x_i \in Y)]\!]$, $[\![\xi_1^{-1}(j)]\!]$ (from both Alice and Bob), and $\xi_1^{-1}(N+i)$ (from Bob). The garbled circuit outputs k_i to Alice, where $k_i = \xi_1^{-1}(j)$ if $\mathrm{Ind}(x_i \in Y) = 1$, and $k_i = \xi_1^{-1}(N+i)$ otherwise. Among $\{k_i\}_{i=1}^B$, $|X \cap Y|$ are $\xi_1^{-1}(j)$ for different $j \in [N]$, while the rest are $\xi_1^{-1}(N+i)$ for different $i \in [B]$. Recall that ξ_1 is a random permutation from $[N+B]$ to $[N+B]$, so these values are distinct numbers drawn from $[N+B]$ uniformly at random without replacement, which carry no information about Bob's data.

Finally, they use another OEP to permute the shares from $\{[\![z'_j]\!]\}_{j=1}^{N+B}$ to $\{[\![z''_i]\!]\}_{i=1}^B$, using the permutation function $\xi_2 : [B] \to [N+B]$ where $\xi_2(i) = k_i$. Note that $z''_i = z'_{k_i} = z_{\xi_1(k_i)}$. When $x_i = y_j$ for some $y_j \in Y$, $z_{\xi_1(k_i)} = z_{\xi_1(\xi_1^{-1}(j))} = z_j$, otherwise $z_{\xi_1(k_i)} = z_{\xi_1(\xi_1^{-1}(N+i))} = z_{N+i} = [\![z_{N+i}]\!]_1 + [\![z_{N+i}]\!]_2 = 0$. We have thus obtained the required payloads in shared form, as desired.

6 OBLIVIOUS YANNAKAKIS

Now we are ready to describe our oblivious protocols for projection-aggregation, semijoin, and join, which form the building blocks of the Yannakakis algorithm. In order to assemble them together, the oblivious protocol of each relational operator must meet the following requirements:

(1) Each input relation is held by either Alice or Bob.
(2) The output relation will be held by one party, say Alice. The tuples in the output relation can only depend on Alice's input relations and the query results.
(3) The annotations of the input and output relations are held by Alice and Bob in shared form.
(4) The transcript of the protocol does not leak any private information. In addition to protecting each party's input relations and shares, we must ensure that
  (a) the size of the output relation only depends on public information (input relation sizes and query result size), not the actual input tuples; and
  (b) the access pattern for each input and output tuples (and their annotations) is indistinguishable.

Next, we present our protocol for each of the relational operators.

6.1 Oblivious Projection-Aggregation

The Yannakakis algorithm requires two different projection-aggregation operators. The first one is $\pi_F^{\oplus}(R)$, while the second one is $\pi_F^{1}(R)$. We first provide a protocol to obliviously compute the former, and then discuss how to modify it for handling the latter.

Computing $\pi_F^{\oplus}(R)$. Suppose Alice holds an annotated relation R of size N, and she would like to compute $\pi_F^{\oplus}(R)$. Note that since Alice has R, she can easily compute the projection $\pi_F(R)$. The challenge is computing the aggregates. Note that the annotations of R are given in shared form, and the aggregates must also be returned in shared form as well. Below, we describe our protocol for computing $\pi_F^{\oplus}(R)$, which uses $\tilde{O}(N)$ communication and running time, and it can be done in a constant number of rounds.

First, Alice locally sorts the tuples in R by F, so that tuples with the same value on F are consecutive. Then, Alice and Bob use OEP to permute the shares of the annotations so that they are consistent with the sorted tuples. Then the idea is to simply add up the annotations one by one, while resetting the sum to 0 whenever a new value on F is encountered. To make this algorithm oblivious, we use a garbled circuit.

Let $\{t_i\}_{i=1}^N$ be the tuples after sorting, and let $v(t_i)$ be the annotation of t_i. They build a garbled circuit with $N-1$ *merge gates*. The inputs to the i-th merge gate include $\mathrm{Ind}(t_i.F = t_{i+1}.F)$ (from Alice), $[\![v(t_{i+1})]\!]$ (from both Alice and Bob), and $[\![z_i]\!]$, which is an output from the $(i-1)$-th gate (except that $z_1 = v(t_1)$). The gate then computes two outputs (in the shared form):

$$z'_i = (1 - \mathrm{Ind}(t_i.F = t_{i+1}.F)) \cdot ([\![z_i]\!]_1 + [\![z_i]\!]_2),$$
$$z_{i+1} = (\mathrm{Ind}(t_i.F = t_{i+1}.F) \cdot z_i) \oplus ([\![v(t_{i+1}]\!]_1 + [\![v(t_{i+1}]\!]_2).$$

Consider all tuples with a particular value on F, say $t_i, \dots, t_j$. It should be clear that z'_j is the $\oplus$-aggregate of their annotations (except that if $j = N$, the aggregate is z_N), while $z'_i = \cdots = z'_{j-1} = 0$. Alice knows this fact, but in order to hide the size and access patterns of the output relation, Alice will put all tuples into the output relation. More precisely, she will put $t_j.F$ into the output relation with annotation $[\![z'_j]\!]$ (or $[\![z_N]\!]$ if $j = N$); for each t_k, $k = i, \dots, j-1$, she will put a dummy tuple into the output relation with annotation $[\![z'_j]\!]$. Thus, technically speaking the output relation is not $\pi_F^{\oplus}(R)$, but one that is semantically equivalent to $\pi_F^{\oplus}(R)$, as all dummy tuples have annotation $[\![0]\!]$. Note that Bob does not know which tuples are dummy since he only has his shares of the annotations.

It is obvious that the circuit has size $O(N)$, so it takes $\tilde{O}(N)$ time and communication to evaluate. Its depth is also $O(N)$, but the number of rounds needed is always a constant, regardless of the depth of the garbled circuit [13].

Computing $\pi_F^1(R)$. Now consider $\pi_F^1(R)$, where the annotations of R are given in shared form. Recall that $\pi_F^1(R) = \pi_F(\{t \in R \mid v(t) \neq 0\})$, while the annotations of all tuples in $\pi_F^1(R)$ are set to 1. However, we cannot let Alice know the relation $\pi_F^1(R)$, which depends on the annotations of R. Instead, we will return an output relation that is semantically equivalent to $\pi_F^1(R)$ to Alice. The output relation contains all tuples in $\pi_F(R)$. For a tuple $t \in \pi_F^1(R)$, its annotation in the output relation will be $[\![1]\!]$; all other tuples will have annotation $[\![0]\!]$. This is consistent with the definition as zero annotation has no contribute to the aggregates. Besides, to hide the output size from Bob, Alice also pads some dummy tuples to the output relation so that it has N tuples.

We modify the aggregation protocol above to compute such a semantically equivalent $\pi_F^1(R)$, as follows. First, we still sort R by F and permute the shares accordingly. Then we build the $N-1$ merge gates as before. However, the input to each merge gate will be $[\![\mathrm{Ind}(v(t_i) \neq 0)]\!]$, which can be computed by another garbled circuit. Meanwhile, in the merge gate, we replace the semiring addition $\oplus$ with $\vee$ (logic `OR`). It can be verified that, in this way, the protocol above indeed computes a semantically equivalent $\pi_F^1(R)$ of size N, as desired.

6.2 Oblivious Semijoin

The Yannakakis algorithm uses two types of semijoins. The first type is actually an annotated join $R = R_F \bowtie^{\otimes} R_{F'}$ but with the constraint $F' \subseteq F$, which is used in the reduce step. The second type is an annotated semijoin $R = R_F \ltimes^{\otimes} R_{F'}$ with no constraints on F and F', which is used in the semijoin step. We first show how to solve the first type obliviously, then the second type can be solved easily.

Computing $R = R_F \bowtie^{\otimes} R_{F'}$. Suppose Alice holds R_F, Bob holds $R_{F'}$, and Alice will also hold the output relation R. The annotations of R_F and $R_{F'}$ are shared, and the annotations of R should also be obtained in shared form. Since the tuples in the output relation R cannot depend on $R_{F'}$, tuples in R_F that cannot join with $R_{F'}$ should not be eliminated; instead, we set their annotations to $[\![0]\!]$. For a tuple $t^{(1)} \in R_F$ that can join with some $t^{(2)} \in R_{F'}$, its annotation should be $[\![v(t^{(1)}) \otimes v(t^{(2)})]\!]$. Thus, R will have the same set of tuples as R_1, but with new annotations. All the new annotations will also be obtained in shared form so that no party knows their actual values; in particular, no one knows which tuples in R_F can or cannot join with $R_{F'}$. Suppose the sizes of R_F and $R_{F'}$ are M and N, respectively. Below we provide a protocol that runs in constant rounds with $\tilde{O}(M+N)$ running time and communication cost.

First, Alice locally computes $X = \pi_{F'}(R_F)$. Then she pads X with dummy tuples so that X still has M tuples. Bob's input is $Y = R_{F'}$. Then they run PSI with secret-shared payloads on X and Y, where the payloads of Y are their annotations in $R_{F'}$. Let x_i be the item in the i-th bin in Alice's cuckoo hash table. Recall that the PSI protocol will return $[\![z_i]\!]$, where $z_i = v(t^{(2)})$ for some $t^{(2)} \in R_{F'}$ that can join with x_i; if such a $t^{(2)}$ does not exist, $z_i = 0$. Next, Alice defines an extended permutation $\xi : [M] \to [B]$ as follows. For each tuple $t_j^{(1)} \in R_1$, if $t_j.F'$ falls into the i-th bin, then $\xi(j) = i$. Then they use OEP to permute these shares $\{[\![z_i]\!]\}$ according to ξ, so that for each $j \in [M]$, Alice and Bob have $[\![z'_j]\!]$, where $z'_j = z_{\xi(j)} = z_i$. Then they use M garbled circuits to compute $[\![v(t_j^{(1)}) \otimes z'_j]\!]$ as the new annotation of $t_j^{(1)}$. This is the desired output, since if $t_j^{(1)}$ can join with some $t^{(2)} \in R_{F'}$, then $z'_j = z_i = v(t^{(2)})$; otherwise $z'_j = z_i = 0$.

Computing $R = R_F \ltimes^{\otimes} R_{F'}$. By definition, $R = R_F \bowtie^{\otimes} \pi^1_{F \cap F'}(R_{F'})$. Recall that $\pi^1_{F \cap F'}(R_{F'})$ denotes the projection of the nonzero-annotated tuples in $R_{F'}$, while setting all their annotations to 1. First they compute $\pi^1_{F \cap F'}(R_{F'})$ by the oblivious projection-aggregation protocol. Then they run the protocol above. Note that the output relation R will have the same set of tuples as R_F, but with possibly different annotations. More precisely, for any tuple $t \in R_F$ that can join with at least one nonzero-annotated tuple in $R_{F'}$, its annotation in R is the same as that in R_F; otherwise its annotation in R is set to $[\![0]\!]$.

When R_F and $R_{F'}$ are held by the same party. The protocol above assumes that R_F and $R_{F'}$ are held by different parties. If they are held by the same party, say Alice (the annotations are still secret-shared between Alice and Bob), then the protocol can be simplified. There is no need to run PSI. First, Alice adds a dummy tuple to $R_{F'}$. Then she locally permutes $R_{F'}$ to obtain a list of $(t^{(1)}, t^{(2)})$ pairs, where $t^{(2)}$ is the tuple in $R_{F'}$ that joins with $t^{(1)}$; if such a $t^{(2)}$ does not exist, Alice sets $t^{(2)}$ to the dummy tuple. This list thus replaces Alice's cuckoo hash table. Then as before they use OEP to permute the shares of $R_{F'}$ to be consistent with the list. Note that the zero annotation of the dummy tuple is refreshed to shares by OEP, so Bob learns nothing from it. Finally, they use a garbled circuit to compute the annotation $[\![v(t^{(1)}) \otimes v(t^{(2)})]\!]$ for each $(t^{(1)}, t^{(2)})$ pair. Note that even if $t^{(2)}$ is a dummy tuple and Alice thus knows that the resulting annotation will be 0, she still has to evaluate the garbled circuit with Bob, so as to hide the access pattern of the output relation. For the annotated semijoin $R = R_F \ltimes R_{F'}$, if Alice has both R_F and $R_{F'}$, then as before we rewrite the semijoin as $R = R_F \bowtie^{\otimes} \pi^{1}_{F\cap F'}(R_{F'})$, and then run this simplified protocol.

6.3 Oblivious Join

In the oblivious join problem, Alice and Bob jointly compute an annotated join $\mathcal{J} = \bowtie^{\otimes}_{F\in\mathcal{E}} R_F$ defined by an acyclic hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Each relation R_F is possessed by either Alice or Bob, with annotations shared. We require that all dangling tuples must have annotation $[\![0]\!]$, which do not contribute to the query result. For any relation R, we use R^* to denote the set of nonzero-annotated tuples in R. As output, the protocol will return $\mathcal{J}^*$ to Alice, with annotations obtained in shared form. Besides, the size $|\mathcal{J}^*|$ is also outputted to Bob, which is allowed as mentioned in Section 4. Unlike our oblivious protocols for projection-aggregation and semijoin, the tuples in output relation of an oblivious join depend on both Alice's and Bob's input relations. Therefore, it can only be used as the last operator in a query plan, so that the query results include $\mathcal{J}^*$, which can therefore be revealed.

Below, we present our oblivious join protocol. It runs in constant rounds with $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$ running time and communication cost, where $\mathrm{IN} = \sum_{F\in\mathcal{E}} |R_F|$ and $\mathrm{OUT} = |\mathcal{J}^*|$.

Our protocol runs in three steps:

(1) *Reveal.* By our assumption on the dangling and non-dangling tuples' annotations, we have $R^*_F = \pi_F(\mathcal{J}^*)$ for every relation R_F. This implies that R^*_F (but not its annotations) can be derived from $\mathcal{J}^*$, so it can be revealed to Alice. Therefore, we use $|R_F|$ garbled circuits to check whether $v(t) = [\![v(t)]\!]_1 + [\![v(t)]\!]_2 = 0$ for $t \in R_F$, and return a dummy tuple (if the answer is "yes") or t (if the answer is "no") to Alice. If t is not dummy, Alice puts it into R^*_F. The running time and communication cost of this step are $\tilde{O}(\mathrm{IN})$.

(2) *Join.* Now for any relation R_F, Alice knows R^*_F. She can then locally compute the join $\mathcal{J}^* = \bowtie_{F\in\mathcal{E}} R^*_F$ using the non-annotated Yannakakis algorithm, and sends $\mathrm{OUT} = |\mathcal{J}^*|$ to Bob. If Alice does not want Bob to learn the exact value of OUT, she may pad dummy tuples to $\mathcal{J}^*$, and send the size of $\mathcal{J}^*$ after padding. The step takes time $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$, but the communication cost is just a constant.

(3) *Compute annotations.* We still need to compute the annotations of $\mathcal{J}^*$ in shared form. Let t_i be the i-th tuple of $\mathcal{J}^*$. For each relation R_F, Alice defines an extended permutation $\xi_F : [\mathrm{OUT}] \to [|R_F|]$ where $\xi_F(i)$ equals to the index of $\pi_F(t_i)$ in R_F, for $i \in [\mathrm{OUT}]$. Then Alice and Bob use OEP to permute the annotations of R_F, so that they learn the shares $\{[\![v(\pi_F(t_i))]\!]\}$. Finally, for each $t_i \in \mathcal{J}^*$, they compute its annotation $[\![v(t_i)]\!] = [\![\otimes_{F\in\mathcal{E}} v(\pi_F(t_i))]\!]$ using a garbled circuit. This step has time and communication cost $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$.

6.4 The Oblivious Yannakakis Algorithm

With all the building blocks in place, we are ready to describe how to make the Yannakakis algorithm oblivious, as follows.

(1) *Reduce.* In the reduce step, the algorithm makes a bottom-up pass over the join tree $\mathcal{T}$, while performing the update $R_{F_p} \leftarrow R_{F_p} \bowtie^{\otimes} \pi^{\oplus}_{F'}(R_F)$ where $F' \subseteq F_p$. This is further decomposed into two steps: computing $\pi^{\oplus}_{F'}(R_F)$ and then the semijoin. The former can be computed by our oblivious projection-aggregation protocol, while the latter by oblivious semijoin. Recall that our oblivious protocols do not change the size of R_{F_p}, but only its annotations. Therefore, the total cost of this step is $\tilde{O}(\mathrm{IN})$.

(2) *Semijoin.* The original Yannakakis algorithm uses two passes of semijoins to remove all dangling tuples. This is not oblivious. Instead, we set their annotations to $[\![0]\!]$. This is exactly what our oblivious semijoin protocol does. Note that our protocol effectively also treats a non-dangling tuple with a 0 annotation as a dangling tuple. This is not an issue, since such a tuple will not produce any nonzero-annotated join results in the third phase anyway. The total cost of this step is also $\tilde{O}(\mathrm{IN})$.

(3) *Full Join.* After the previous phases, only output attributes remain and the annotations of dangling tuples have been set to $[\![0]\!]$. We can then invoke our oblivious join protocol to compute the full join results $\mathcal{J}^*$. Recall that the oblivious join protocol computes the annotations of $\mathcal{J}^*$ in shared form, but we can just reveal these annotations to Alice, as they are part of the query results. The cost of this step is $\tilde{O}(\mathrm{IN} + \mathrm{OUT})$.

Remark. In fact, we could also make the original two-phase Yannakakis algorithm oblivious. However, doing oblivious semijoins before the *Reduce* phase would incur unnecessary computation involving relations that should have been reduced.

6.5 Optimizations

In this section, we introduce how to improve our protocol in some cases when they have more public information.

When a party has the relation and its annotations. In most cases, each of the input relation to the Yannakakis algorithm is fully known by a party, including its annotations. In this case, the annotated projection-aggregation can be directly computed locally, and in the oblivious semijoin protocol, when the annotations of Bob's relation are fully known by Bob, they only need to run the PSI protocol with payloads instead of the secret-shared version. In particular, sometimes the annotations are the same and public, e.g. computing the join-aggregate query size (`count` aggregation), where the annotations of each input relation are all 1. In this case, the oblivious semijoin protocol even degenerates to a simple PSI protocol. Note that this optimization usually only works at the start of the Yannakakis protocol, as for each protocol, the annotations of the output relation becomes in shared form.

When a party holds a subtree containing the root node. Suppose the relations that belong to Alice form a connected part that contains the root node in $\mathcal{T}$. In the bottom-up reduce step of Yannakakis algorithm, first all computations are locally done on Bob's relations, and then they perform some oblivious semijoins, where all the annotations of Bob's relations are known by Bob, so we only need to use PSI with payloads. Afterwards, all computations are on Alice's relations, although the annotations are shared. Hence we only need the simplified oblivious semijoin protocol. In a word, in this special type of queries, we do not need to use PSI with secret-shared payloads protocol, so the efficiency can be improved.

7 EXTENSIONS

Cyclic and non-free-connex queries. Our oblivious Yannakakis protocol achieves $\tilde{O}(\text{IN} + \text{OUT})$ cost when the query is acyclic and free-connex. If the query is not, we can invoke the *generalized hypertree decomposition (GHD)* framework [15, 17] to turn it into one, and then run our protocol. The resulting cost, however, will be a polynomial whose exponent depends on the width of the GHD. We omit the details from the conference version of the paper.

Selection conditions. Suppose we are given a join-aggregate query where there is a selection condition ϕ_F on each input relation R_F. Then we have the following options, depending on the privacy requirement.

(1) If the selectivity of a condition ϕ_F is not private, then we can simply replace R_F with $\sigma_{\phi_F}(R_F)$ when running the oblivious Yannakakis algorithm. In this case, the input size IN only includes $\sigma_{\phi_F}(R_F)$, and the cost of the algorithm will be lower.
(2) If the selectivity of a condition ϕ_F is private, then we replace all tuples in R_F that do not satisfy ϕ_F with dummy tuples, and then run the oblivious Yannakakis algorithm. The cost does not decrease even though the query is only interested in a subset of the tuples of R_F. This is actually unavoidable, since if the cost were reduced, the cost itself would reveal information about the selectivity of ϕ_F.
(3) If the precise selectivity is private, but it is alright to reveal some upper bound, then we can replace R_F with $\sigma_{\phi_F}(R_F)$, and then add some dummy tuples. This strikes a good balance between cost and privacy, and is perhaps a common scenario in practice. In Example 1.1, suppose there is a selection condition on R_1 that selects only customers in a particular state. It is probably alright to reveal the total number of customers in that state, or at least an upper bound.

Query composition. Some aggregation queries are not free-connex join-aggregate queries by our precise definition, but they can be decomposed into two or more such queries. For example, suppose we replace `sum` with `avg` in the query of Example 1.1, then there is no semiring that can make it into one join-aggregate query per se, but obviously it suffices to compute the `sum` and `count` for each `class`, both of which are free-connex join-aggregate queries. However, since the `sum` and `count` are *not* in the final query results, we cannot compute them out and do a division in plaintext. Fortunately, the oblivious Yannakakis algorithm only outputs the join results to Alice, with annotations obtained in shared form. Thus, we first run two instances of the oblivious Yannakakis algorithm to compute the `sum` and `count` in shared form for each `class`. Then, we use a garbled circuit for each `class` to compute the `avg`, and only reveal the `avg` to Alice. Query 8 and 9 in the experiment section also provide examples of query decomposition.

Protecting privacy against query results. By definition of our 2PC model, Alice will learn the query results. If the query results are sensitive, then one can add noise following the theory of *differential privacy* [12] as mentioned in Section 1. A widely used approach is to first compute some measure of *sensitivity* Δ of an aggregate in the query results, and then add to the aggregate a noise drawn from the Laplace distribution with parameter Δ/ε, where ε is the *privacy parameter*. This approach can be easily incorporated into our protocol. Recently, Johnson et al. [18] proposed a simple measure of sensitivity for join-count queries, which only depends on the maximum frequency of attribute values in each relation. Thus, we just need Alice and Bob to find the maximum frequencies, and then compute Δ using an $\tilde{O}(1)$-size garbled circuit. Finally, Bob generates a random noise from the Laplace distribution and adds it to the query result using another garbled circuit, before revealing the result to Alice.

However, for join-aggregate queries where the aggregation function is not `count`, how to calculate a meaningful sensitivity measure Δ is still an open problem. Nevertheless, any such measure can be incorporated into our protocol, provided that Δ can be computed by a circuit.

8 EXPERIMENTS

8.1 Queries

We tested with the following queries from the TPC-H benchmark. We allow the set of relations to be arbitrarily partitioned between Alice and Bob.

Query 3. This query is already a free-connex join-aggregate query in its vanilla form:

```
select o_orderkey, o_orderdate, o_shippriority,
  sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem
where c_mktsegment = 'AUTOMOBILE'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '1995-03-13'
  and l_shipdate > date '1995-03-13'
group by o_orderkey, o_orderdate, o_shippriority;
```

We consider the selectivities of all the selection conditions to be private. So during preprocessing, we replaced all the tuples not satisfying these conditions with dummy tuples. The annotations of `lineitem` are `l_extendedprice*(1-l_discount)`, while they are all 1 for other relations, except for dummy tuples. Note that after the *Reduce* step in oblivious Yannakakis algorithm, the join tree has only one node. Therefore we can simply reveal nonzero-annotated tuples of the relation in this node, without going through the *Semijoin* and *Full Join* steps.

Query 10. This query illustrates a case where the cost can be reduced if some relations are public. As the parties can agree on a common mapping between nations' names and their keys, the `nation` relation can be considered as public knowledge. This way, the query can be simplified to the following one:

```
select c_custkey, c_name, c_nationkey,
  sum(l_extendedprice * (1 - l_discount)) revenue
from customer, orders, lineitem
where c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= date '1993-08-01'
  and o_orderdate < date '1993-11-01'
  and l_returnflag = 'R'
group by c_custkey, c_name, c_nationkey;
```

The original query has `n_name` as an output attribute instead of `c_nationkey`. However, after obtaining the query results, the receiver can easily look up the `n_name` from the `nation` relation.

Query 18. This query has a subquery in its where clause:

```
select c_name,c_custkey,o_orderkey,o_orderdate,
  o_totalprice,sum(l_quantity)
from customer, orders, lineitem
where o_orderkey in (select l_orderkey
    from lineitem
    group by l_orderkey
    having sum(l_quantity) > 300)
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by c_name,c_custkey,
  o_orderkey,o_orderdate,o_totalprice;
```

Note that the subquery can be evaluated locally by the party that possesses `lineitem`. However, in order to hide the result size of the subquery, we need to add dummy tuples so that its size is the same as the original `lineitem` relation.

Query 8. First, similar to Query 10, we assume `nation` and `region` are public knowledge. We remove them from the join, and rewrite the selection conditions on nations' and regions' names so that they refer to `s_nationkey` or `c_nationkey` instead:

```
select o_year, sum(case
  when s_nationkey = 8 then volume
  else 0 end) / sum(volume) as mkt_share
from (select s_nationkey,
    extract(year from o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount) as volume
  from part, supplier, lineitem, orders, customer
  where p_partkey = l_partkey
    and s_suppkey = l_suppkey
    and l_orderkey = o_orderkey
    and o_custkey = c_custkey
    and c_nationkey in (8,9,12,18,21)
    and o_orderdate between
      date '1995-01-01' and date '1996-12-31'
    and p_type = 'SMALL PLATED COPPER') as all_nations
group by o_year;
```

Although this query contains a subquery, its purpose is merely to extract `o_year` from `o_orderdate`. By treating `o_year` as a virtual column, this query is a join followed by an aggregation. However, the aggregation to be computed is the ratio between two sums, so there is no semigroup that can yield this aggregation directly. Nevertheless, it can be composed into two join-aggregate queries as described in Section 7. We compute the two sum aggregates, in shared form, for every year. Each sum is a join-aggregate query. The two queries use different annotations for `supplier`: The first query uses Ind(`s_nationkey=8`) where Ind is the indicator function, and the second one uses 1 for all tuples. Finally, we use a garbled circuit to compute the ratio of the two sums for each year.

Query 9. As before, we first remove `nation` from the query:

```
select s_nationkey,o_year,sum(amount)
from(
  select s_nationkey,
    extract(year from o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount)
      - ps_supplycost * l_quantity as amount
  from part,supplier,lineitem,partsupp,orders
  where s_suppkey = l_suppkey
    and ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
    and p_name like '%green%'
  ) as profit
group by s_nationkey, o_year;
```

This is an acyclic join-aggregate query, but not free-connex, because its two output attributes `s_nationkey` and `o_year` cannot be put at the top of any join tree. Although we could have used the GHD framework to convert it to a free-connex query, there is a simpler way to get around, by exploiting the fact that `nation` is public, and `s_nationkey` has a small domain size of 25. We thus decompose this query into 25 queries, each corresponding to one particular `s_nationkey`. For each such query, we remove `s_nationkey` from the `group by`, and add a selection condition enforcing `s_nationkey` to be that particular nation. Furthermore, the query has a complicated aggregation function that cannot be evaluated by a single join-aggregate query, but, as in Query 8, we can decompose it into two aggregates: The first computes `sum(l_extendedprice*(1-l_discount))` while the second computes `sum(ps_supplycost*l_quantity)`. Then they locally do subtractions on their corresponding shares of annotations, and then reveal the results to Alice.

8.2 Experiment Setup

We implemented the oblivious Yannakakis protocol for the 5 queries above with manually written code in C++. For benchmarking, we measured the running time and communication cost in the non-private setting, for which we simply run the query using MySQL. The communication cost for the non-private setting is set to the input size. We would have liked to compare with SMCQL, but we have not been able to run queries other than the given examples using their code[5], while none of their examples has joins with more than two relations. Therefore, we wrote a garbled circuit on our own to just compute the Cartesian product of the relations and apply join conditions on it, while ignoring all other operators. Thus, the actual cost of SMCQL evaluating the full query can only be higher. For example, our garbled circuit computing the join of the 3 relations in Query 3, consisting a total of 7,655 tuples, took 2.8

[5]https://github.com/smcql/smcql

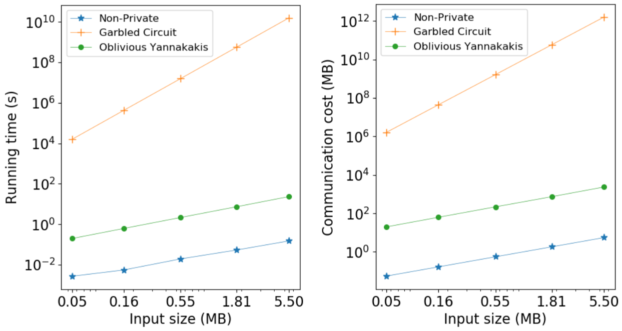

Figure 2: The time and cost of Query 3

hours, while SMCQL reportedly took a single day to run a query involving two relations with hundreds of tuples [5].

We used the TPC-H data generator to generate 5 datasets of sizes 1MB, 3MB, 10MB, 33MB, and 100MB, respectively. Note that an oblivious protocol is designed to have indistinguishable behaviours on different inputs, so the actual tuples in the relations do not matter, except the relation sizes.

We use parameter values suggested in the security literature: The computational security parameter κ is set to 128, the statistical security parameter is $\sigma = 40$, and the bit length of all attributes is $\ell = 16$. The running times are measured on a laptop with an Intel Core i5-7200U CPU running a single thread. They are CPU times and do not include the time for communication (which would depend on the network bandwidth).

8.3 Experimental Results

Figure 2-6 show the results. Note that both the x-axis and y-axis are in log-scale. The input size is equal to the total size of the columns involved in the query (effective size). For obvious reasons, we could not run the garbled circuit except on the smallest dataset, so the results on larger datasets are extrapolated. This is actually very accurate, since the cost is proportional to the size of the circuit, which we know exactly.

There is really no surprise in the results, as oblivious Yannakakis has been proved to have costs linear in the input size. Nevertheless, it is still mind-boggling to see the concrete numbers: On the 100M dataset (effective data size is 5M to 8M), the garbled circuit for Query 3 would take 300 years, sending 1 EB of data around, while these numbers are 20 seconds and 1 GB for oblivious Yannakakis. On Query 9, which is the most complicated query involving 5 relations, 2 aggregations, and 25 subqueries, the difference is between the end of the Universe[6] and time for a cup of tea.

9 FUTURE WORK

We are currently working on a query compiler to generate code from SQL automatically. We are also working on supporting more SQL features, such as distinct count, theta-joins, and integration with differential privacy. Hardening our protocol against malicious adversaries and supporting more than two parties are also interesting directions to look into.

[6]Many theoretical physicists believe the Universe will end between 2.8 billion years and 22 billion years from now.

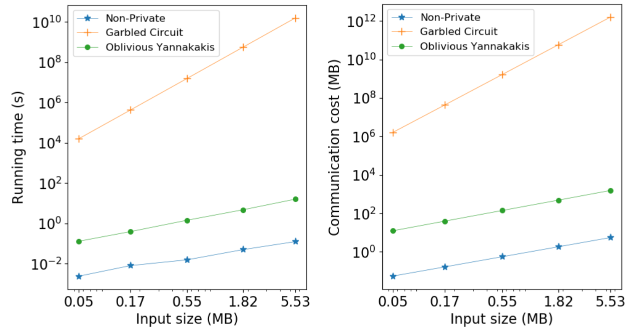

Figure 3: The time and cost of Query 10

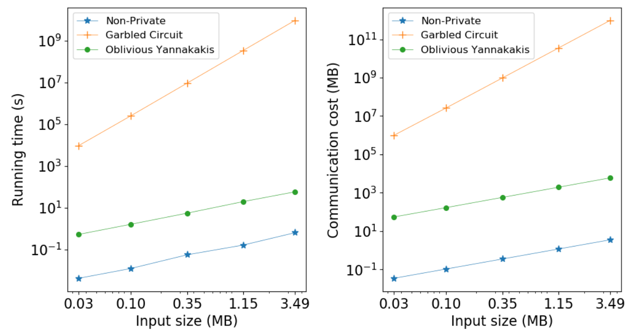

Figure 4: The time and cost of Query 18

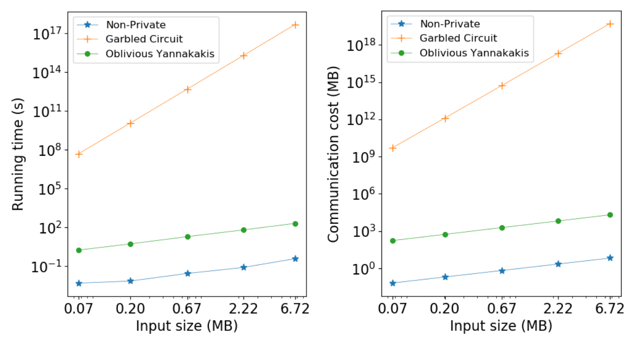

Figure 5: The time and cost of Query 8

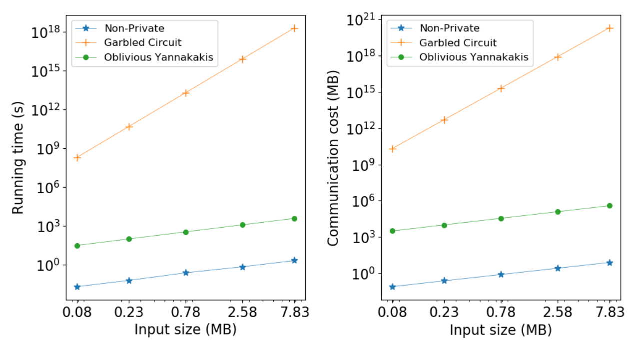

Figure 6: The time and cost of Query 9

REFERENCES

[1] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. 2005. Two Can Keep A Secret: A Distributed Architecture for Secure Database Services. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 186–199. http://cidrdb.org/cidr2005/papers/P16.pdf

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1033–1036. https://doi.org/10.1145/2463676.2467797

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.

[4] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: A Trusted Hardware based Database with Privacy and Data Confidentiality. In *SIGMOD*.

[5] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure querying for federated databases. *Proceedings of the VLDB Endowment* 10, 6 (2017), 673–684.

[6] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Efficient SQL query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018), 307–320.

[7] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 169–188.

[8] David Chaum. 1984. Blind signature system. In *Advances in cryptology*. Springer, 153–153.

[9] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1243–1255.

[10] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*. Springer, 643–662.

[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY: A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[12] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.

[13] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246.

[14] Oded Goldreich, S. Micali, and Avi Wigderson. 1987. How to play ANY mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 218–229. https://doi.org/10.1145/28395.28420

[15] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. System Sci.* 64, 3 (2002), 579–627.

[16] Dennis Heimbigner and Dennis McLeod. 1985. A federated architecture for information management. *ACM Transactions on Information Systems (TOIS)* 3, 3 (1985), 253–278.

[17] 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*. 91–106.

[18] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[19] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From oblivious AES to efficient and secure database join in the multiparty setting. In *International Conference on Applied Cryptography and Network Security*. Springer, 84–101.

[20] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-efficient oblivious database manipulation. In *International Conference on Information Security*. Springer, 262–277.

[21] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[22] Payman Mohassel and Saeed Sadeghian. 2013. How to hide circuits in MPC an efficient framework for private function evaluation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 557–574.

[23] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[24] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium (USENIX Security 15)*. 515–530.

[25] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient circuit-based psi with linear communication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 122–153.

[26] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 797–812. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas

[27] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 85–100.

[28] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: a secure database using SGX. In *IEEE Security & Privacy*.

[29] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *PVLDB*.

[30] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

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

[32] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167.