Oblivious Query Processing

Arvind Arasu
Microsoft Research
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

ABSTRACT

Motivated by cloud security concerns, there is an increasing interest in database systems that can store and support queries over encrypted data. A common architecture for such systems is to use a *trusted* component such as a cryptographic co-processor for query processing that is used to securely decrypt data and perform computations in plaintext. The trusted component has limited memory, so most of the (input and intermediate) data is kept encrypted in an *untrusted* storage and moved to the trusted component on "demand."

In this setting, even with strong encryption, the data access pattern from untrusted storage has the potential to reveal sensitive information; indeed, all existing systems that use a trusted component for query processing over encrypted data have this vulnerability. In this paper, we undertake the first formal study of *secure query processing*, where an adversary having full knowledge of the query (text) and observing the query execution learns nothing about the underlying database other than the result size of the query on the database. We introduce a simpler notion, *oblivious query processing*, and show formally that a query admits secure query processing *iff* it admits oblivious query processing. We present oblivious query processing algorithms for a rich class of database queries involving selections, joins, grouping and aggregation. For queries not handled by our algorithms, we provide some initial evidence that designing oblivious (and therefore secure) algorithms would be hard via reductions from two simple, well-studied problems that are generally believed to be hard. Our study of oblivious query processing also reveals interesting connections to database join theory.

1. INTRODUCTION

There is a trend towards moving database functionality to the cloud and many cloud providers have a *database-as-a-service (DbaaS)* offering [2, 21]. A DbaaS allows an application to store its database in the cloud and run queries over it. Moving a database to the cloud, while providing well-documented advantages [8], introduces data security concerns [28]. Any data stored on a cloud machine is potentially accessible to snooping administrators and to attackers who gain illegal access to cloud systems. There have been well-known instances of data security breaches arising from such adversaries [30].

A simple mechanism to address these security concerns is encryption. By keeping data stored in the cloud encrypted we can thwart the kinds of attacks mentioned above. However encryption makes computation and in particular *query processing* over data difficult. Standard encryption schemes are designed to "hide" data while we need to "see" data to perform computations over it. Addressing these challenges and designing database systems that support query processing over encrypted data is an active area of research [4, 6, 14, 27, 32] and industry effort [22, 23].

A common architecture [4, 6] for query processing over encrypted data involves using *trusted hardware* such as a cryptographic co-processor [16], designed to be inaccessible to an adversary. The trusted hardware has access to the encryption key, some computational capabilities, and limited storage. During query processing, encrypted data is moved to the trusted hardware, decrypted, and computed on. The trusted hardware has limited storage so it is infeasible to store within it the entire input database or intermediate results generated during query processing; these are typically stored encrypted in an *untrusted* storage and moved to the trusted hardware only when necessary. Other approaches to query processing over encrypted data rely on (partial) *homomorphic encryption* [14, 27, 32] or using the client as the trusted module[1] [14, 32]. These approaches have limitations in terms of the class of queries they can handle or data shipping costs they incur (see Section 7), and they are not the main focus of this paper.

The systems that use trusted hardware currently provide an *operational* security guarantee that any data outside of trusted hardware is encrypted [4, 6]. However, this operational guarantee does not translate to *end-to-end data security* since, even with strong encryption, the data movement patterns to and from trusted hardware can potentially reveal information about the underlying data. We call such information leakage *dynamic information leakage* and we illustrate it using a simple join example.

EXAMPLE 1. *Consider a database with tables* `Patient(PatId,Name,City)` *and* `Visit(PatId,Date,Doctor)` *storing patient details and their doctor visits. These tables are encrypted by encrypting each record using a standard encryption scheme and stored in untrusted memory. Using suitably strong encryption*[2]*, we can ensure that the adversary does not learn anything from the encrypted tables other than their sizes. (Such an encryption scheme is non-deterministic so two encryptions of the same record would look seemingly unrelated.)*

Consider a query that joins these two tables on `PatId` *column*

[1] All of our results hold for this setting, but they are less interesting.
[2] And padding to mask record sizes.

10.5441/002/icdt.2014.07

using a nested loop join algorithm. The algorithm moves each patient record to the trusted hardware where it is decrypted. For each patient record, all records of `Visit` *table are moved one after the other to the trusted hardware and decrypted. Whenever the current patient record p and visit record v have the same* `PatId` *value, the join record* $\langle p, v\rangle$ *is encrypted and produced as output. An adversary observing the sequence of records being moved in and out of trusted hardware learns the join-graph. For example, if* 5 *output records are produced in the time interval between the first and second* `Patient` *record moving to the trusted hardware, the adversary learns that some patient had* 5 *doctor visits. We can show that similar information leakage occurs for other standard join algorithms such as hash join and sort-merge join.*

The above discussion raises the natural question whether we can design query processing algorithms that avoid such dynamic information leakage and provide end-to-end data security. The focus of this paper is to seek an answer to this question; in particular, as a contribution of this paper, we formalize a strong notion of (end-to-end) secure query processing, develop efficient and secure query processing algorithms for a large class of queries, and discuss why queries outside of this class are unlikely to have efficient secure algorithms.

There exists an extensive body of work on *Oblivious RAM (ORAM) Simulation* [10, 12, 29, 33], a general technique that makes memory accesses of an arbitrary program appear random by continuously shuffling memory and adding spurious accesses. In Example 1, with ORAM simulation the data accesses would appear random to an adversary and we can show that the adversary learns no information other than the total number of data accesses.

Given general ORAM simulation, why design specialized secure query processing algorithms? We defer a full discussion of this issue to Section 1.2, but for a brief motivation consider sorting an encrypted array of size n. Just as in Example 1, the data access patterns of a standard sorting algorithm such as quicksort reveals information about the underlying data. An ORAM simulation of quicksort would hide the access patterns; indeed, the adversary does not even learn that a sort operation is being performed. However, with current state-of-the-art ORAM algorithms, it would incur an overhead of $\Theta(\log^2 n)$ per access[3] of the original algorithm making the overall complexity of sorting $\Theta(n \log^3 n)$. Instead, we could exploit the semantics of sorting and design a secure sorting algorithm that has the (optimal) time complexity of $\Theta(n \log n)$ [11]. Here the adversary does learn that the operation being performed is sorting (but does not learn anything about the input being sorted) but we get significant performance benefits. As we show in the rest of the paper, designing specialized secure query processing algorithms helps us gain similar performance advantages over generic ORAM simulations.

The exploration in this paper is part of the *Cipherbase* project [7], a larger effort to design and prototype a comprehensive database system, relying on specialized hardware for storing and processing encrypted data in the cloud.

1.1 Overview of Contributions

Secure Query Processing: Informally, we define a query processing algorithm for a query Q to be *secure* if an adversary having full knowledge of the text of Q and observing the execution of Q does not learn anything about the underlying database D other than the result size of Q over D. The query execution happens within a *trusted module (TM)* not accessible to the adversary. The input database and possibly intermediate results generated during query execution are stored (encrypted) in an *untrusted memory (UM)*. The adversary has access to the untrusted memory and in particular can observe the sequence of memory locations accessed and data values read and written during query execution. Throughout, we assume a *passive* adversary, who does not actively interfere with query processing.

When defining security, we grant the adversary knowledge of query Q which ensures that data security does not depend on the query being kept secret. Databases are typically accessed through applications and it is often easy to guess the query from a knowledge of the application. Our formal definition of secure query processing (Section 2.2) generalizes the informal definition above and incorporates query security in addition to data security. Our formal definition relies on machinery from standard cryptography such as *indistinguishability experiments*, but there are some subtleties specific to database systems and applications that we capture; Appendix A discusses these issues in greater detail.

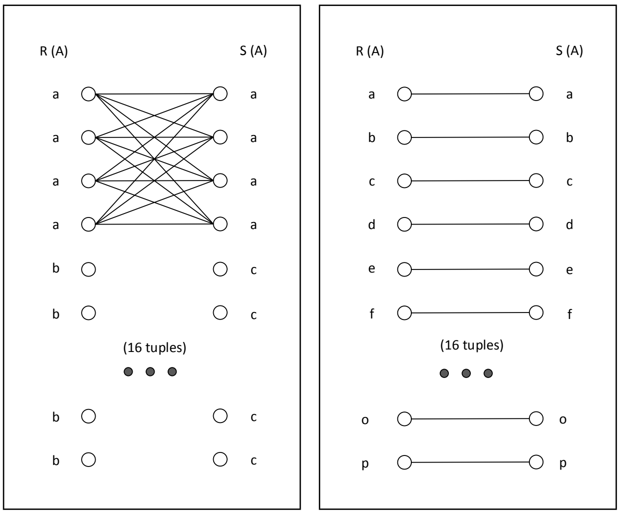
We note that simply communicating the (encrypted) query result over an untrusted network reveals the result size, so a stronger notion of security seems impractical in a cloud setting. Also, our definition of secure query processing implies that an adversary with an access to the cloud server gets no advantage over an adversary who can observe only the communications between the client and the database server, assuming both of them have knowledge of the query.

Oblivious Query Processing Algorithms: Central to the idea of secure query processing is the notion of *oblivious query processing*. Informally, a query processing algorithm is oblivious if its (untrusted) memory access pattern is independent of the database contents once we fix the query and its input and output sizes. We can easily argue that any secure algorithm is oblivious: otherwise, the algorithm has different memory access patterns for different database instances, so the adversary learns something about the database instance by observing the memory access pattern of the algorithm. Interestingly, obliviousness is also a *sufficient* condition in the sense that any oblivious algorithm can be made secure using standard cryptography (Theorem 3). The idea of reducing security to memory access obliviousness was originally proposed in [10] for general programs in the context of software protection. Note that obliviousness is defined with respect to memory accesses to the untrusted store; any memory accesses internal to TM are invisible to the adversary and do not affect security.

Our challenge therefore is to design oblivious algorithms for database queries. We seek oblivious algorithms that have small TM memory footprint since all practical realizations of the trusted module such as cryptographic co-processors have limited storage (few MBs) [16]. Without this restriction a simple oblivious algorithm is to read the entire database into TM and perform query processing completely within TM.

To illustrate challenges in designing oblivious query processing algorithms consider the simple join query $R(A, \ldots) \bowtie S(A, \ldots)$ which seeks all pairs of tuples from R and S that agree on attribute A. Figure 1 shows two instances for this join, represented as a binary "join-graph". Each R and S tuple is shown as a vertex and its attribute A value is shown adjacent to it (lower case letters $a, b, \ldots$). An edge exists between an R tuple and an S tuple having the same value in attribute A, and each edge represents a join output. We note that both instances have the same input output characteristics, $|R| = |S| = |R \bowtie S| = 16$, so an oblivious algorithm is required to have the same memory access pattern for both instances. However, the internal structure of the join graph is greatly different. All "natural" join algorithms that use sorting or hashing to bring together joinable tuples are sensitive to the join graph structure and

[3] Assuming "small" $\text{polylog}(n)$ trusted memory.



(a) Instance 1

(b) Instance 2

Figure 1: Two join instances with same input/output sizes

therefore not oblivious.

Traditional database query processing is *non-oblivious* for two reasons: First, traditional query processing proceeds by identifying a query plan, which is a tree of operators with input tables at the leaves. The operators are (conceptually) evaluated in a bottom-up fashion and the output of each operator forms an input of its parent. In some cases, this bottom-up evaluation can be pipelined. In others, the output of an operator needs to be generated fully before the parent can consume it, and such intermediate output needs to be temporarily stored in untrusted memory. This renders the overall query processing non-oblivious since the size of the intermediate output can vary depending on the database instance, even if we fix input and output sizes. Second, standard implementations of database operators such as filters, joins, and grouping are not oblivious, so even if the query plan consisted of a single operator, the resulting query processing algorithm would not be oblivious. In summary, traditional query processing is non-oblivious at both inter- and intra-operator levels, and we need to fundamentally rethink query processing to make it oblivious.

Our first main algorithmic contribution is that a surprisingly rich class of database queries admit *efficient* oblivious algorithms (Sections 3, 4 and 5).

THEOREM 1. *(Informal) There exists an oblivious (secure) query processing algorithm that requires $O(\log n)$ storage in TM for any database query involving joins, grouping aggregation (as the outermost operation), and filters, if (1) the non-foreign key join graph is acyclic and (2) all grouping attributes are connected through foreign key joins, where n denotes the sum of query input and output sizes. Further, assuming no auxiliary structures, the running time of the algorithm is within $O(\log n)$ (multiplicative factor) of the running time of the best insecure algorithm.*

Theorem 1 suggests an interesting connection between secure query processing and database join theory since *acyclic joins* are a class of join queries known to be tractable [24]. We note that the class of queries is fairly broad and representative of real-world analytical queries. For example, most queries in the well-known TPC-H benchmark [31] belong to this class, i.e., admit efficient secure algorithms.

Assuming no auxiliary structures such as indexes, our algorithms are efficient and within $O(\log n)$ of the running time of the best insecure algorithm. While the no-index condition makes our results less relevant for transactional workloads, where indexes play an important role, they are quite relevant for analytical workloads where indexes play a less critical role. In fact, query processing in emerging database architectures such as *column stores* [1] has limited or no dependence on indexes.

Further, with minor modifications to our algorithms using the oblivious external memory sort algorithm of [13], we get oblivious and secure algorithms with excellent external (untrusted) memory characteristics.

THEOREM 2. *(Informal) For the class of queries in Theorem 1 there exists an oblivious (secure) algorithm with I/O complexity within multiplicative factor $\log_{M/B}(n/B)$ of that of the optimal insecure algorithm, where B is the block size and M is the TM memory.*

In particular, if we have $\Omega(\sqrt{n})$ memory in TM our external memory algorithms perform a constant number of scans to evaluate the queries they handle.

Interestingly, for the special case of joins, secure algorithms have been studied in the context of privacy preserving data integration [19]. The algorithm proposed in [19] proceeds by computing a cross product of the input relations followed by a (secure) filter. Our algorithms are significantly more efficient and handle grouping and aggregation.

Negative Results: We have reason to believe that queries outside of the class specified in Theorem 1 do not admit secure efficient algorithms. We show that the existence of secure algorithms would imply more efficient algorithms for variants of classic hard problems such as 3SUM (Section 6). These hardness arguments suggest that we must accept a weaker notion of security if we wish to support a larger class of queries.

1.2 Oblivious RAM Simulations

ORAM simulations first proposed by Goldreich and Ostrovsky [10] is a general technique for making memory accesses oblivious that works for arbitrary programs. Specifically, ORAM simulation is the online transformation of an arbitrary program P to an equivalent program P' whose memory accesses appear random (more precisely, drawn from some distribution that depends only on the number of memory accesses of P). By running P' within a *secure CPU (TM)* and using suitable encryption, an adversary observing the sequence of memory accesses to an untrusted memory learns nothing about P and its data other than its number of memory accesses. Current ORAM simulation techniques work by adding a virtualization layer that continuously shuffles (untrusted) memory contents and adds spurious memory accesses, so that the resulting access pattern looks random.

A natural idea for oblivious query processing, implemented in a recent system [20], would be to run a standard query processing algorithm under ORAM simulation. However, the resulting query processing is *not* secure for our definition of security since it reveals more than just the output size. ORAM simulation, since it is designed for general programs, does not hide the total number of memory accesses; in the context of standard query processing, this reveals the size of intermediate results in a query plan. Understanding the utility of this weaker notion of security in the context of database systems is an interesting direction of future work.

For database queries that admit polynomial time algorithms (which includes queries covered by Theorem 1) we can design oblivious algorithms based on ORAM simulation: the number of memory accesses of such an algorithm is bounded by some polynomial[4] $p(n, m)$, where n is the input size, and m, the output size. We modify the algorithm with dummy memory accesses so that the number of memory accesses for any instance with input size n and output size m is exactly $p(n, m)$. An ORAM simulation of the modified algorithm is oblivious. We note that we need to precisely specify $p(n, m)$ upto constants (not asymptotically), otherwise the number of memory accesses would be slightly different for different (n, m) instances making the overall algorithm non-oblivious. In practice, working out a precise upper-bound $p(n, m)$ for arbitrarily complex queries is a non-trivial undertaking.

Our algorithms which are designed to exploit the structure and semantics of queries have significant performance benefits over the ORAM-based technique sketched above given the current state-of-the-art in ORAM simulation. For simplicity, assume for this discussion that the query output size $m = O(n)$. For small TM memory (polylog(n)), the current best ORAM simulation techniques [29, 18] incur an overhead of $\Theta(\log^2 n)$ memory accesses per memory access of the original algorithm. This implies that the time complexity of any ORAM-based query processing algorithm is lower-bounded by $\Omega(n \log^2 n)$. In contrast, our algorithms have a time complexity of $O(n \log n)$ and use $O(\log n)$ TM memory.

Also, by construction ORAM simulation randomly sprays memory accesses and destroys locality of reference, reducing effectiveness of caching and prefetching in a memory hierarchy. In a disk setting, a majority of memory accesses of ORAM simulation result in a random disk seek and we can show that any ORAM-based query processing algorithm incurs $\Omega(\frac{n}{B \log M} \log^2 \frac{n}{B})$ disk seeks, where M denotes the size of TM memory. In contrast, all of our algorithms are scan-based except for the oblivious sorting, which incurs $O(\log_{M/B}(n/B)) \cdot o(n/B)$ seeks.

2. PROBLEM FORMULATION

2.1 Database Preliminaries

A *relation schema*, $R(\bar{A})$, consists of a relation symbol R and associated attributes $\bar{A} = (A_1, \ldots, A_k)$; we use $\mathit{Attr}(R)$ to denote the set of attributes $\{A_1, \ldots, A_k\}$ of R. An attribute A_i has an associated set of values called its *domain*, denoted $\mathcal{D}(A_i)$. We use $\mathcal{D}(R)$ to denote $\mathcal{D}(\bar{A}) = \mathcal{D}(A_1) \times \ldots \times \mathcal{D}(A_k)$. A *database schema* is a set of relation schemas $R_1, \ldots, R_m$. A (relation) instance corresponding to schema $R(A_1, \ldots, A_k)$ is a bag (multiset) of *tuples* of the form $\langle a_1, \ldots, a_k \rangle$ where each $a_i \in \mathcal{D}(A_i)$. A database instance is a set of relation instances. In the following we abuse notation and use the term relation (resp. database) to denote both relation schema and instance (resp. database schema and instance). We sometimes refer to relations as *tables* and attributes as *columns*.

Given a tuple $t \in R$ and an attribute $A \in \mathit{Attr}(R)$, $t[A]$ denotes the value of the tuple on attribute A; as a generalization of this notation, if $\mathcal{A} \subseteq \mathit{Attr}(R)$ is a set of attributes, $t[\mathcal{A}]$ denotes the tuple t restricted to attributes in $\mathcal{A}$.

We consider two classes of database queries. A *select-project-join (SPJ)* query is of the form $\pi_{\mathcal{A}}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$, where the projection π is duplicate preserving (we use multiset semantics for all queries) and $\bowtie$ refers to the natural join. For $R_1 \bowtie R_2$, each tuple $t_1 \in R_1$ joins with each tuple $t_2 \in R_2$ such that $t_1[\mathit{Attr}(R_1) \cap \mathit{Attr}(R_2)] = t_2[\mathit{Attr}(R_1) \cap \mathit{Attr}(R_2)]$ to produce an output tuple t over attributes $\mathit{Attr}(R_1) \cup \mathit{Attr}(R_2)$ that agrees with t_1 on attributes $\mathit{Attr}(R_1)$ and with t_2 on attributes $\mathit{Attr}(R_2)$. The second class of queries involves grouping and aggregation and is of the form $\mathbb{G}_{\mathcal{G}}^{F(A)}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$ and we call such queries GSPJ queries. Given relation R, $\mathcal{G} \subseteq \mathit{Attr}(R)$, $A \in \mathit{Attr}(R)$, $\mathbb{G}_{\mathcal{G}}^{F(A)}(R)$, represents grouping by attributes in $\mathcal{G}$ and computing aggregation function F over attribute A.

2.2 Secure Query Processing

A *relation encryption scheme* is used to encrypt relations. It is a triple of polynomial algorithms (Enc, Dec, Gen) where Gen takes a security parameter k and returns a key K; Enc takes a key K, a plaintext relation instance R and returns a ciphertext relation $\mathcal{C}_R$; Dec takes a ciphertext relation $\mathcal{C}_R$ and key K and returns plaintext relation R if K was the key under which $\mathcal{C}_R$ was produced. A relation encryption scheme is also a database encryption scheme: to encrypt a database instance we simply encrypt each relation in the database.

Informally, a relation encryption scheme is *IND-CPA secure* if a polynomial time adversary with access to encryption oracle cannot distinguish between the encryption of two instances $R^{(1)}$ and $R^{(2)}$ of relation schema R such that $|R^{(1)}| = |R^{(2)}|$ ($|R^{(1)}|$ denotes the number of tuples in $R^{(1)}$). Assuming all tuples of a given schema have the same representational length (or can be made so using padding), we can construct IND-CPA secure relation encryption by encrypting each tuple using a standard encryption scheme such as AES in CBC mode (which is believed to be IND-CPA secure for message encryption). The detail that encryption is at a tuple-granularity is relevant for our algorithms which assume that we can read and decrypt one tuple at a time.

Our formal definition of secure query processing captures: (1) Database security: An adversary with knowledge of a query does not learn anything other than the result size of the query by observing query execution; (2) Query security: An adversary without knowledge of the query does not learn the constants in the query from query execution. Appendix A contains a discussion of query security.

A *query template* $\mathcal{Q}$ is a set of queries that differ only in constants. An example template is the set $\{\sigma_{A=1}(R), \sigma_{A=2}(R), \cdots\}$ which we denote $\sigma_{A=*}(R)$.

A query processing algorithm $\mathbb{A}_{\mathcal{Q}}$ for a query template $\mathcal{Q}$ takes as input an encrypted database instance $\mathsf{Enc}_K(D)$, a query $Q \in \mathcal{Q}$ and produces as output $\mathsf{Enc}_K(Q(D))$; Algorithm $\mathbb{A}_{\mathcal{Q}}$ has access to encryption key K and the encryption scheme is IND-CPA secure. Our goal is to make algorithm $\mathbb{A}_{\mathcal{Q}}$ secure against a passive adversary who observes its execution. Algorithm $\mathbb{A}_{\mathcal{Q}}$ runs within the trusted module TM. The TM also has a small amount of internal storage invisible to the adversary. Algorithm $\mathbb{A}_{\mathcal{Q}}$ has access to a large amount of untrusted storage which is sufficient to store $\mathsf{Enc}_K(D)$ and any intermediate state required by $\mathbb{A}_{\mathcal{Q}}$. The *trace* of an execution of algorithm $\mathbb{A}_{\mathcal{Q}}$ is the sequence of untrusted memory accesses $\mathit{read}(i)$ and $\mathit{write}(i, \mathit{value})$, where i denotes the memory location.

We define security of algorithm $\mathbb{A}_{\mathcal{Q}}$ using the following *indistinguishability* experiment:

1. Pick $K \leftarrow \mathsf{Gen}(1^k)$
2. The adversary $\mathcal{A}$ picks two queries $Q_1 \in \mathcal{Q}$, $Q_2 \in \mathcal{Q}$ with the same template and two database instances $D^{(1)} = \{R_1^{(1)}, \ldots, R_n^{(1)}\}$ and $D^{(2)} = \{R_1^{(2)}, \ldots, R_n^{(2)}\}$ having the same schema such that (1) $|R_i^{(1)}| = |R_i^{(2)}|$ for all $i \in [1, n]$; and (2) $|Q_1(D^{(1)})| = |Q_2(D^{(2)})|$.
3. Pick a random bit $b \leftarrow \{0, 1\}$ and let τ_b denote the trace of $\mathbb{A}_{\mathcal{Q}}(\mathsf{Enc}_K(D^{(b)}), Q_b)$.
4. The adversary $\mathcal{A}$ outputs prediction b' given τ_b, $\mathsf{Enc}_K(D^{(b)})$,

[4] This argument does not depend on p being a polynomial, any function works.

and $\mathsf{Enc}_K(Q_b(D^{(b)}))$.

We say adversary $\mathcal{A}$ succeeds if $b' = b$. Algorithm $\mathbb{A}_\mathcal{Q}$ is secure if for any polynomial time adversary $\mathcal{A}$, the probability of success is at most $1/2 + negl(k)$ for some negligible function[5] $negl$. We note that our definition of security captures both database security, since an adversary can pick $Q_1 = Q_2$, and query security, since an adversary can pick $D^{(1)} = D^{(2)}$.

2.3 Oblivious Query Processing

As discussed in Section 1, oblivious query processing is a simpler notion that is equivalent to secure query processing. Fix an algorithm $\mathbb{A}_\mathcal{Q}$. For an input $I = \mathsf{Enc}_K(D)$ and query $Q \in \mathcal{Q}$, the memory access sequence $\mathcal{M}_{\mathbb{A}_\mathcal{Q}}(I, Q)$ is the sequence of UM memory reads $r(i)$ and writes $w(i)$, where i denotes the memory location; the value being read/written is not part of $\mathcal{M}_{\mathbb{A}_\mathcal{Q}}(I, Q)$. In general, $\mathbb{A}_\mathcal{Q}$ is randomized and $\mathcal{M}_{\mathbb{A}_\mathcal{Q}}(I, Q)$ is a random variable defined over all possible memory access sequences. Algorithm $\mathbb{A}_\mathcal{Q}$ is oblivious if the distribution of its memory access sequences is independent of database contents once we fix the query output and database size. Formally, Algorithm $\mathbb{A}_\mathcal{Q}$ is oblivious if for any memory access sequence M, any two queries $Q_1, Q_2 \in \mathcal{Q}$, any two database encryptions $I_1 = \mathsf{Enc}_{K_1}(D^{(1)})$, $I_2 = \mathsf{Enc}_{K_2}(D^{(2)})$:

$$\Pr[\mathcal{M}_{\mathbb{A}_\mathcal{Q}}(I_1, Q_1) = M] = \Pr[\mathcal{M}_{\mathbb{A}_\mathcal{Q}}(I_2, Q_2) = M]$$

where $D^{(1)} = \{R_1^{(1)}, \ldots, R_n^{(1)}\}$ and $D^{(2)} = \{R_1^{(2)}, \ldots, R_n^{(2)}\}$ have the same schema and: (1) $|R_i^{(1)}| = |R_i^{(2)}|$ for all $i \in [1, n]$; and (2) $|Q_1(D^{(1)})| = |Q_2(D^{(2)})|$.

Our definition of obliviousness is more stringent than the one used in ORAM simulation. In ORAM simulation, the memory access distribution can depend on the total number of memory accesses, while our definition precludes dependence on the total number of memory accesses once the query input and output sizes are fixed. The following theorem establishes the connection between oblivious and secure query processing.

THEOREM 3. *Assuming one-way functions exist, the existence of an oblivious algorithm for a query template $\mathcal{Q}$ implies the existence of a secure algorithm for $\mathcal{Q}$ with the same asymptotic performance characteristics (TM memory required, running time).*

The idea of using obliviousness to derive security from access pattern leakage was originally proposed in [10] and the proof of Theorem 3 is similar to the proof of analogous Theorem 3.1.1 in [10]. Informally, we get secure query processing by ensuring both data security of values stored in untrusted memory and access pattern obliviousness. Data security can be achieved by using encryption, and secure encryption schemes exist assuming the existence of one-way functions. It follows that the existence of oblivious query processing algorithms implies the existence of secure algorithms. Based on Theorem 3, the rest of the paper focuses on oblivious query processing and does not directly deal with encryption and data security.

3. INTUITION

This section presents a high level intuition behind our algorithms. Consider the binary join $R(A, \ldots) \bowtie S(A, \ldots)$ and the join graph instance shown in Figure 2(a). Lower case letters a, b, represent values of the joining column A; ignore the subscripts on a and b for now. We add identifiers r_1-r_3 and s_1-s_4 to tuples so that we can refer to them in text.

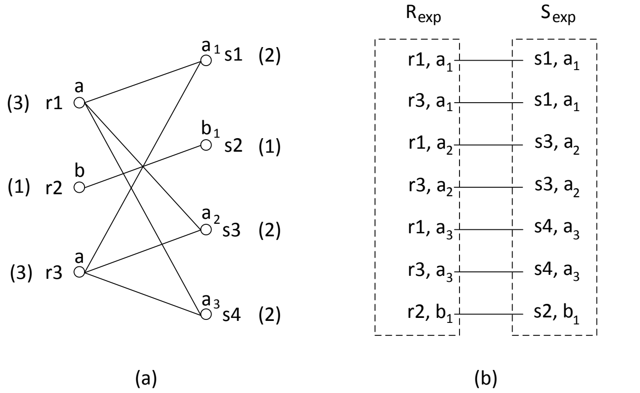


Figure 2: Illustration of Oblivious Binary Join

Our oblivious binary join algorithm works in two stages: In the first stage, we compute the contribution of each tuple to the final output. This is simply the degree of the tuple in the join graph; this value is shown within parenthesis in Figure 2(a). For example, the degree of r_1 is 3, and degree of r_2, 1. In the second stage, we *expand* R to R_{exp} by duplicating each tuple as many times as its degree; r_1 occurs 3 times in R_{exp}, r_2 once, and so on. We similarly, expand S to S_{exp}. The expansions R_{exp} and S_{exp} are shown within boxed rectangles in Figure 2(b). The final join output is produced by "stitching" together R_{exp} and S_{exp} as illustrated in Figure 2(b). The expansions R_{exp} and S_{exp} are *sequences* whose ordering is picked carefully to ensure that stitching the ith tuple in R_{exp} with the ith tuple in S_{exp} indeed produces the correct join output.

A central component of the above algorithm are oblivious implementations of two simple primitives that we call *semi-join aggregation* and *expansion*. Semi-join aggregation computes the degree of each tuple in a join and expansion expands a relation by duplicating each tuple a certain number of times such as its degree.

The same approach generalizes to multiway joins if the overall query is *acyclic* [24]. Informally, to compute $R \bowtie S \bowtie T$, we would compute the contribution of each tuple to the *final* join output and use these values to expand input relations to R_{exp}, S_{exp}, and T_{exp}, which are then stitched together to produce the final join output.

4. PRIMITIVES

This section introduces a few core primitives and presents oblivious algorithms—algorithms that have the same UM memory access patterns once we fix input and output sizes—for these primitives. These primitives serve two purposes: First, as discussed in Section 3, they are building blocks for our oblivious query processing algorithms; Second, they introduce notation to help us concisely specify our algorithms, and reason about their obliviousness and performance.

There exist oblivious algorithms for all the primitives of this section having time complexity $O(n \log n)$ and requiring $O(\log n)$ TM memory, where n denotes the sum of input and output sizes. Some of these algorithms rely on an oblivious sort; an optimal $O(n \log n)$ oblivious sort algorithm that uses $O(1)$ TM memory is presented in [11]. Due to space constraints we defer presenting oblivious algorithms for the simpler primitives to the full version of the paper [5].

[5] A negligible function $negl(k)$ is a function that grows slower than $\frac{1}{p(k)}$ for any polynomial $p(k)$; e.g., $\frac{1}{2^k}$.

	A
r_1	1
r_2	2
r_3	1

R

	A	B
s_1	1	1
s_2	2	1
s_3	2	1

S

A	B	C	D	E
1	2	1	2	1
2	4	1	6	2
1	2	2	8	1

$\tilde{R}$

Figure 3: Illustration of primitives: $\tilde{R} \leftarrow R.(B \leftarrow 2A).(C \leftarrow \text{ID}_A).(D \leftarrow \text{RSum}(B)).(E \overset{\ltimes}{\leftarrow} Sum(S.B))$. r_1-r_3 and s_1-s_3 are names we use to refer to the tuples.

Relation Augmentation: This primitive adds a derived column to a relation. In the simplest form the derived column is obtained by applying a function to existing columns; many primitives we introduce subsequently are more complex instantiations of relation augmentation. We use the notation $R.(A \leftarrow \mathit{Func})$ to represent relation augmentation which adds a new derived column A using some function Func and produces an output relation with schema $\mathit{Attr}(R) \cup \{A\}$. For example, $R.(B \leftarrow 2A)$ adds a new column B whose value is twice that of A (see Figure 3). Our notation allows composition to be expressed more concisely; e.g., $R.(B \leftarrow 2A).(F \leftarrow A + B)$.

Grouping Identity: This relation augmentation primitive adds a new *identity* column within a group; identity column values are of the form $1, 2, \ldots$. In particular, we use the notation $R.(A \leftarrow \text{ID}^{\mathcal{O}}_{\mathcal{G}})$ where $\mathcal{G} \subseteq \mathit{Attr}(R)$ is a set of grouping columns and $\mathcal{O} \subseteq \mathit{Attr}(R)$ is a set of ordering columns. To get the output, we partition the tuples by the grouping columns $\mathcal{G}$, order the tuples within each partition by $\mathcal{O}$, and assign ids based on this ordering. (We break ties arbitrarily, so the output can be non-deterministic.) $\mathcal{G}$ and $\mathcal{O}$ can be empty and omitted. For example, $R.(\mathit{Id} \leftarrow \text{ID})$ assigns an unique id to each record in R. In Figure 3, for $R.(C \leftarrow \text{ID}_A)$, we partition by A, so r_1 and r_3 go to the same partition; tuple r_1 gets a C value of 1, and r_3, a C value of 2.

Grouping Running Sum: This primitive is a generalization of grouping identity and adds a running sum column to a relation. It is represented $R.(A \leftarrow \text{RSum}^{\mathcal{O}}_{\mathcal{G}}(B))$; it groups a relation by $\mathcal{G}$ and orders tuples in a group by $\mathcal{O}$ and stores the running sum of B column values in a new column A. In particular, grouping identity $R.(\mathit{Id} \leftarrow \text{ID}^{\mathcal{O}}_{\mathcal{G}})$ can be expressed as $R.(X \leftarrow 1).(\mathit{Id} \leftarrow \text{RSum}^{\mathcal{O}}_{\mathcal{G}}(X))$. See Figure 3 for an example.

Generalized Union: A *generalized union* of R and S, denoted $R \dot{\cup} S$, produces a relation with schema $\mathit{Attr}(R) \cup \mathit{Attr}(S)$ that contains tuples from both R and S. Tuples of R have a *null* value for attributes in $\mathit{Attr}(S) - \mathit{Attr}(R)$, and those of S, a null value for attributes in $\mathit{Attr}(R) - \mathit{Attr}(S)$.

Sequences: Sorting and Stitching: Although the inputs and outputs of our algorithms are relations represented as *sequences*, the ordering is often unimportant and we mostly do not emphasize the sequentiality. We use the notation $\langle R \rangle$ to represent some sequence corresponding to R. When a particular ordering is desired, we represent the ordering as $\langle R \rangle_{\mathcal{O}}$ where $\mathcal{O} \subseteq \mathit{Attr}(R)$ denote the ordering attributes.

One operation on sequences that cannot be represented over bags is "stitching" two sequences of the same length (see Figure 2(b) for an example): Given two sequences $\langle R \rangle$ and $\langle S \rangle$ of the same length n, the operation $\langle R \rangle \cdot \langle S \rangle$ produces a sequence of length n with schema $\mathit{Attr}(R) \cup \mathit{Attr}(S)$ and the ith tuple of the sequence is obtained by concatenating the ith tuple of $\langle R \rangle$ and the ith tuple of $\langle S \rangle$; we ensure when invoking this operation that the ith tuples of both sequences agree on $\mathit{Attr}(R) \cap \mathit{Attr}(S)$ if the intersection is nonempty.

Algorithm 1 Semi-Join Aggregation: $R.(X \overset{\ltimes}{\leftarrow} Sum(S.Y))$

```
1: procedure SEMIJOINAGG($R, S, X, Y$)
2:     $\tilde{R} \leftarrow R.(Src \leftarrow 1).(Y \leftarrow 0)$
3:     $\tilde{S} \leftarrow S.(Src \leftarrow 0)$
4:     $U \leftarrow \tilde{R} \dot{\cup} \tilde{S}$
5:     $U \leftarrow U.(X \leftarrow \text{RSum}^{Src}_{Attr(R) \cap Attr(S)}(Y))$
6:     Output $\pi_{Attr(R),X}(\sigma_{Src=1}(U))$
7: end procedure
```

Filters: Consider the filter $\sigma_P(R)$. The simple algorithm that scans each tuple $t \in R$, checks if it satisfies P, and outputs it if does, is *not* oblivious. (E.g., simply reordering tuples in R changes the memory write pattern.)

The oblivious sorting algorithm can be used to design a simple oblivious algorithm for selection (filter). To evaluate $\sigma_P(R)$, we sort R such that tuples that satisfy predicate P occur before tuples that do not. We scan the sorted table and output the tuples that satisfy P and stop when we encounter the first tuple that does not satisfy P. The overall data access pattern depends only on input and output sizes and is therefore oblivious.

4.1 Semi-Join Aggregation

Semi-join aggregation, denoted $R.(A \overset{\ltimes}{\leftarrow} Sum(S.B))$, is equivalent[6] to the relational algebra expression $\mathbb{G}^{A \leftarrow \text{SUM}(S.B)}_{Attr(R)}(R \bowtie S)$. This operation adds a new derived column A; for each tuple $t_R \in R$, we obtain value of A by identifying all $t_S \in S$ that join with t_R (agree on all common attributes $\mathit{Attr}(R) \cap \mathit{Attr}(S)$) and summing over $t_S[B]$ values. As discussed in Section 3, we introduce this primitive to compute the degree of a tuple in a join graph. In particular, the degree of each R tuple in $R \bowtie S$ is obtained by $\tilde{S} \leftarrow S.(X \leftarrow 1)$, $R.(\mathit{Degree} \overset{\ltimes}{\leftarrow} Sum(\tilde{S}.X))$. In Figure 3, r_2 joins with two tuples s_2 and s_3, so $r_2[E]$ is $s_2[B] + s_3[B] = 2$.

Oblivious Algorithm: Algorithm 1 presents an oblivious algorithm for semi-join aggregation $R.(X \overset{\ltimes}{\leftarrow} Sum(S.Y))$. (In all our algorithms, each step involves one of our primitives and is implemented using the oblivious algorithm for the primitive.) It adds a "lineage" column Src in Steps 2 and 3; the value of Src is set to 1 for all R tuples and 0 for all S tuples. A Y column initialized to 0 is added to all R tuples. Step 4 computes a generalized union U of $\tilde{R}$ and $\tilde{S}$. Adding the running sum within each $\mathit{Attr}(R) \cap \mathit{Attr}(S)$ group adds the required aggregation value into each R tuple (Step 5); the running sum computation is ordered by Src to ensure that all S tuples within an $\mathit{Attr}(R) \cap \mathit{Attr}(S)$ group occur before the R tuples. Finally, the oblivious filter $\sigma_{Src=1}$ in Step 6 extracts the R tuples from U. Figure 4 shows the intermediate tables generated by Algorithm 1 for sample tables $R(\mathit{Id}, A)$ and $S(A, Y)$.

THEOREM 4. *Algorithm 1 obliviously computes semi-join aggregation $R.(X \overset{\ltimes}{\leftarrow} Sum(S.Y))$ of two tables in $O((n_R + n_S)\log(n_R + n_S))$ time and using $O(1)$ TM memory, where $n_R = |R|$ and $n_S = |S|$ denote the input table sizes.*

PROOF. (Sketch) For each step of Algorithm 1 the input and output sizes are one of n_R, n_S, and $n_R + n_S$ and each step is locally oblivious in its input and output sizes. The overall algorithm is therefore oblivious. Further, the oblivious algorithms for each step require $O(1)$ TM memory. □

[6] This equality holds only when R has not duplicates.

Id	*A*	*Y*	*Src*
1	a	0	1
2	b	0	1

(a): $\tilde{R}$

A	*Y*	*Src*
a	2	0
b	3	0
a	4	0

(b): $\tilde{S}$

Id	*A*	*Y*	*Src*	*X*
-	a	2	0	2
-	a	4	0	6
1	a	0	1	6
-	b	3	0	3
2	b	0	1	3

(c): U

Figure 4: Sample computation of $R.(X \overset{\ltimes}{\leftarrow} Sum(S.Y))$

4.2 Expansion

This primitive duplicates each tuple of a relation a number of times as specified in one of the columns. In particular, the output of $\text{Exp}_W(R)$, $W \in Attr(R)$ and $\mathcal{D}(W) \subseteq \mathbb{N}$ is a relation instance with same schema, $Attr(R)$, that has $t[W]$ copies of each tuple $t \in R$. For example, given an instance of $R(A, W) : \{\langle a, 1\rangle, \langle b, 2\rangle\}$, $\text{Exp}_W(R)$ is given by $\{\langle a, 1\rangle, \langle b, 2\rangle, \langle b, 2\rangle\}$. As discussed in Section 3, expansion plays a central role in our join algorithms.

4.2.1 Oblivious Algorithm

We now present an oblivious algorithm to compute $\text{Exp}_W(R)$. For presentational simplicity, we slightly modify the representation of the input. The modified input to the expansion is a sequence of pairs $(\langle r_1, w_1\rangle, \dots, \langle r_n, w_n\rangle)$, where r_is are values (tuples) drawn from some domain and $w_i \in \mathbb{N}$ are non-negative *weights*. The desired output is some sequence containing (in any order) w_i copies of each r_i. We call such a sequence a *weighted sequence*.

The input size of expansion is n and the output size is $m \overset{\text{def}}{=} \sum_{i=1}^{n} w_i$, so memory access pattern of an oblivious algorithm depends on only these two quantities. The naive algorithm that reads each $\langle r_i, w_i\rangle$ into TM and writes out w_i copies of r_i is not oblivious, since the output pattern depends on individual weights w_i.

We first present an oblivious algorithm when the input sequence has a particular property we call *prefix-heavy*; we use this algorithm as a subroutine in the algorithm for the general case.

DEFINITION 5. *A weighted sequence* $(\langle r_1, w_1\rangle, \dots, \langle r_n, w_n\rangle)$ *is* prefix-heavy *if for each* $\ell \in [1, n]$, $\frac{1}{\ell}\sum_{i=1}^{\ell} w_i \geq \frac{1}{n}\sum_{i=1}^{n} w_i$.

The average weight of any prefix of a prefix-heavy sequence is greater-than-or-equal to the overall average weight. Any weighted sequence can be reordered to get a prefix-heavy sequence, e.g., by sorting by non-decreasing weight. The sequence $(\langle a, 4\rangle, \langle b, 1\rangle, \langle c, 2\rangle)$ is prefix-heavy, while $(\langle b, 1\rangle, \langle a, 4\rangle, \langle c, 2\rangle)$ is not.

Algorithm 2 presents an oblivious algorithm for expanding prefix heavy weighted sequences. To expand the sequence $I = (\langle r_1, w_1\rangle, \dots, \langle r_n, w_n\rangle)$, the algorithm proceeds in n (input-output) steps. Let $w_{avg} = (\sum_{i=1}^{n} w_i)/n$ denote the average weight of the sequence. In each step, it reads one weighted record (Step 5) and produces w_{avg} (unweighted) records in the output; the actual number w_{curr} is either $\lfloor w_{avg} \rfloor$ or $\lceil w_{avg} \rceil$, when w_{avg} is fractional (Step 6).

Call a record $\langle r_i, w_i\rangle$ *light* if $w_i \leq w_{avg}$ and *heavy*, otherwise. If the current record $\langle r_i, w_i\rangle$ is light, w_i copies of r_i are produced in the output; if it is heavy, a counter $\mathcal{C}[r_i]$ is initialized with count w_i denoting the number of copies of r_i available for (future) outputs. Previously seen heavy records are used to make up the "balance" and ensure w_{avg} records are produced in each step. The counters $\mathcal{C}$ are internal to TM and are not part of the data access pattern. Figure 5 shows the steps of Algorithm 2 for the sequence $(\langle a, 4\rangle, \langle b, 1\rangle, \langle c, 2\rangle)$.

Algorithm 2 is oblivious since its input-output pattern is fixed once the input size n and output size $m = \sum_{i=1}^{n} w_i$ is fixed. Note that $w_{avg} = m/n$ is fixed once input and output sizes are fixed.

Algorithm 2 Oblivious Expansion of prefix heavy sequences

```
1: procedure EXPANDPREFIXHEAVY($I$)
     Assume: $I = (\langle r_1, w_1\rangle, \dots, \langle r_n, w_n\rangle)$
     Require: $I$ is prefix heavy
2:     $w_{avg} \leftarrow (\sum_{i=1}^{n} w_i)/n$
3:     $\mathcal{C}_{TM} \leftarrow \phi$                          ▷ counters within TM
4:     for $i = 1$ to $n$ do
5:         Read $\langle r_i, w_i\rangle$ to TM.
6:         $w_{curr} \leftarrow \lfloor i \cdot w_{avg} \rfloor - \lfloor (i-1) \cdot w_{avg} \rfloor$
7:         if $w_i \leq w_{curr}$ then
8:             Append $w_i$ copies of $r_i$ to output
9:             $w_{curr} \leftarrow w_{curr} - w_i$
10:        else
11:            $\mathcal{C}_{TM}[r_i] \leftarrow w_i$
12:        end if
13:        while $w_{curr} > 0$ do
14:            $r_j \leftarrow$ argmin$_k$ $r_k$ has a counter in $\mathcal{C}_{TM}$
15:            if $\mathcal{C}_{TM}[r_j] > w_{curr}$ then
16:                Append $w_{curr}$ copies of $r_j$ to output
17:                $\mathcal{C}_{TM}[r_j] \leftarrow \mathcal{C}_{TM}[r_j] - w_{curr}$
18:                $w_{curr} \leftarrow 0$
19:            else
20:                Append $\mathcal{C}_{TM}[r_j]$ copies of $r_j$ to output
21:                $w_{curr} \leftarrow w_{curr} - \mathcal{C}_{TM}[r_j]$
22:                // $\mathcal{C}_{TM}[r_j] \leftarrow 0$
23:                Remove $r_j$ from $\mathcal{C}_{TM}$
24:            end if
25:        end while
26:    endfor
27: end procedure
```

Step	Input	Output	Counters
1	$\langle a, 4\rangle$	a, a	$\mathcal{C}[a] = 2$
2	$\langle b, 1\rangle$	b, a, a	$\mathcal{C} = \phi$
3	$\langle c, 2\rangle$	c, c	$\mathcal{C} = \phi$

Figure 5: Algorithm 2 over sequence $(\langle a, 4\rangle, \langle b, 1\rangle, \langle c, 2\rangle)$

In the worst case, the number of counters maintained by Algorithm 2 can be $\Omega(n)$.

EXAMPLE 2. *Consider the sequence* $w_1 = \cdots = w_{n/4} = 4$ *and* $w_{n/2+1} = \cdots = w_n = 0$. *After reading* $n/4$ *records, we can show that Algorithm 2 requires* $\approx 3n/16$ *counters.*

However, any weighted sequence can be re-ordered so that it is prefix heavy and the number of counters used by Algorithm 2 is $O(1)$ as stated in Lemma 6 and illustrated in the following example.

EXAMPLE 3. *We can reorder the weight sequence in Example 2 as* $\langle 4, 0, 0, 0, 4, 0, 0, 0, \dots, \rangle$ *interleaving* 3 *light records inbetween two heavy records. We can show that Algorithm 2 requires just one counter for this sequence.*

More generally, the basic idea is to interleave sufficient number of light records between two heavy records so that average weight of any prefix is barely above the overall average, which translates to fewer number of counters. In Example 3, we can suppress just one heavy record to make the average weight of any prefix $\leq w_{avg}$. We call such sequences *barely prefix heavy*.

LEMMA 6. *Any weighted sequence* I *can be re-ordered as a prefix heavy sequence* I' *such that Algorithm 2 requires* $O(1)$ *counters to process* I'.

Lemma 6 suggests that we can design a general algorithm for expansion by first reordering the input sequence to be barely prefix

Algorithm 3 Oblivious Expansion $\mathrm{Exp}_W(R)$

```
 1: procedure EXPAND($R, W$)
 2:     $m \leftarrow \mathbb{G}^{\mathrm{SUM}(W)}$                                   ▷ output size
 3:     $\tilde{R} \leftarrow R.(\tilde{W} \leftarrow 2^{\lceil \log_2 W \rceil})$                     ▷ weight rounding
 4:     $\tilde{m} \leftarrow \mathbb{G}^{\mathrm{SUM}(\tilde{W})}$                              ▷ assert: $\tilde{m} < 2m$
 5:     $\tilde{R} \leftarrow \tilde{R} \mathbin{\bar{\cup}} \{\langle dummy \rangle\}.(W \leftarrow 0).(\tilde{W} \leftarrow 2m - \tilde{m})$
 6:     $\mathcal{D}_{TM} \leftarrow \mathbb{G}_{\tilde{W}}^{\mathrm{COUNT}(*)}(\tilde{R})$                 ▷ rounded weight distr
 7:     $\tilde{R} \leftarrow \tilde{R}.(Id \leftarrow \mathrm{ID})$                              ▷ Attach ids
 8:     $\langle \tilde{R}_{bph} \rangle \leftarrow$REORDERBARELYPREFIXHEAVY($\tilde{R}, \tilde{D}_{TM}$)
 9:     $\tilde{R}_{exp} \leftarrow$EXPANDPREFIXHEAVY($\langle \tilde{R}_{bph} \rangle$)
10:     $\tilde{R}_{exp} \leftarrow \tilde{R}_{exp}.(Rank \leftarrow \mathrm{ID}_{Id})$
11:     Output $\pi_{Attr(R)}(\sigma_{Rank <= W}(\tilde{R}_{exp}))$
12: end procedure
```

heavy and using Algorithm 2. The main difficulty lies in *obliviously* reordering the sequence to make it barely prefix heavy. We do not know how to do this directly; instead, we transform the input sequence to a modified sequence by *rounding* weights (upwards) to be a power of 2. We can concisely represent the full rounded weight distribution using logarithmic space. We store this rounded distribution within TM and use it to generate a barely prefix heavy sequence. Details of the algorithm to reorder a sequence to make it barely prefix heavy is presented in the full-version of the paper [5].

Algorithm 3 presents our oblivious expansion algorithm. Step 3 performs weight rounding. Directly expanding with these rounded weights produces a sequence of length $\tilde{m}$; the resulting algorithm would not be oblivious since $\tilde{m}$ does not depend on n and m (output size) alone. We therefore add (Step 5) a dummy tuple with rounded weight $2m - \tilde{m}$ (and actual weight 0). Expanding this table produces $2m$ tuples. Step 6 computes the distribution of rounded weights. We note that this step consumes $\tilde{R}$ and produces no output since $\mathcal{D}_{TM}$ remains within TM. Step 8 reorders the table (sequence) to make it barely prefix heavy which is expanded using Algorithm 2 in Step 9. Steps 10 and 11 filter out dummy tuples produced due to rounding using an oblivious selection algorithm.

THEOREM 7. *Algorithm 3 obliviously expands an input table in time $O((n+m)\log(n+m))$ using $O(\log(n+m))$ TM memory, where n and m denote the input and output sizes, respectively.*

PROOF. (Sketch) The input size of each step is one of n, $n+1$ or m. The output size of each step is one of 1, n, $n+1$, or m. Each step is locally oblivious, so all data access patterns are fixed once we fix n and m. □

5. QUERY PROCESSING ALGORITHMS

We now present oblivious query processing algorithms for SPJ and GSPJ queries. Recall from Section 2.1 that these are of the form $\pi_{\mathcal{A}}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$ and $\mathbb{G}_{\mathcal{G}}^{F(X)}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$. Instead of presenting a single algorithm, we present algorithms for various special cases that together formalize (and prove) the informal characterization in Theorem 1. We begin by presenting in Section 5.1 an oblivious algorithm for binary join. In Section 5.2 we discuss extensions to multiway joins. Section 5.3 discusses grouping and aggregation, Section 5.4 discusses selection predicates, and Section 5.5 discusses how key-foreign key constraints can be exploited.

5.1 Binary Join

Recall the discussion from Section 3 (Figure 2) which presents the high level intuition behind our binary join algorithm: Informally, to compute $R(A,\ldots) \bowtie S(A,\ldots)$ we begin by computing the degree of each tuple of R and S in the join graph; we use semi-join aggregation to compute the degree. We then expand R and S to R_{exp} and S_{exp} by duplicating each tuple of R and S as many times as its degree. By construction, R_{exp} contains the R-half of join tuples and S_{exp} contains the S-half, and we stitch them together to produce the final join output (see Figure 2(b)).

Algorithm 4 Binary Natural Join: $R \bowtie S$

```
 1: procedure BINARYJOIN($R, S$)
 2:     $\mathcal{J} \leftarrow Attr(R) \cap Attr(S)$                          ▷ join attrs
 3:     $\tilde{R} \leftarrow R.(N \leftarrow 1)$                              ▷ tuple multiplicity
 4:     $\tilde{R} \leftarrow \tilde{R}.(Id \leftarrow \mathrm{ID})$                           ▷ Add an id column
 5:     $\tilde{S} \leftarrow S.(N \leftarrow 1)$                              ▷ tuple multiplicity
 6:     $\tilde{S} \leftarrow \tilde{S}.(Id \leftarrow \mathrm{ID})$                           ▷ Add an id column
 7:     $\tilde{R} \leftarrow \tilde{R}.(N_S \overset{\ltimes}{\leftarrow} Sum(S.N))$                   ▷ Compute degree
 8:     $\tilde{S} \leftarrow \tilde{S}.(N_R \overset{\ltimes}{\leftarrow} Sum(R.N))$                   ▷ Compute degree
 9:     $\tilde{S} \leftarrow \tilde{S}.(JId \leftarrow \mathrm{ID}_{\mathcal{J}})$
10:     $R_{exp} \leftarrow \mathrm{Exp}_{N_S}(\tilde{R})$
11:     $R_{exp} \leftarrow R_{exp}.(JId \leftarrow \mathrm{ID}_{Id})$
12:     $S_{exp} \leftarrow \mathrm{Exp}_{N_R}(\tilde{S})$
13:     Output $\pi_{Attr(R) \cup Attr(S)}(\langle R_{exp} \rangle_{\mathcal{J},JId} \cdot \langle S_{exp} \rangle_{\mathcal{J},JId})$
14: end procedure
```

One remaining detail is to order R_{exp} and S_{exp} so that they can be stitched to get the join result. Simply ordering by the join column values does not necessarily produce the correct result. We attach a subscript to join values on the S side so that different occurrences of the same value get a different subscript; the three a values now become a_1, a_2, a_3. We expand S as before remembering the subscripts, so there are two copies each of a_1, a_2, and a_3. We expand R slightly differently: each a tuple on R is expanded 3 times and we produce one copy of each subscript. For example, tuple r_1 is expanded to (r_1, a_1), (r_1, a_2) and (r_1, a_3). Sorting by the subscripted values and stitching produces the correct join result.

Formal Algorithm: Algorithm 4 presents our join algorithm. Steps 7 and 8 compute the join degrees (N_S and N_R, resp) for each R and S tuple using a semi-join aggregation. Step 9 is a grouping identity operation. All S tuples agreeing on join columns $\mathcal{J}$ belong to the same group, and each gets a different identifier. This step plays the role of assigning subscripts to S tuples in Figure 2(a). Steps 10 and 12 expand $\tilde{R}$ and $\tilde{S}$ based on the join degrees. Step 11 is another grouping identity operation. All tuples in R_{exp} that originated from the same R tuple belong to the same group, and each gets a different identifier. This has the effect of expanding R tuples with a different subscript. Step 13 stitches expansions of R and S to get the final join output. Figure 6 illustrates Algorithm 4 for the example shown in Figure 2. Note the correspondence between Jid column values and subscripts in Figure 2.

THEOREM 8. *Algorithm 4 obliviously computes the binary natural join of two tables R and S in time $\Theta(n_R \log n_R + n_S \log n_S + m \log m)$, where $n_R = |R|$, $n_S = |S|$, and $m = |R \bowtie S|$ using $O(\log(n_R + n_S))$ TM memory.*

5.2 Multiway Join

We now consider multiway joins, i.e., natural joins between q relations $R_1 \bowtie \cdots \bowtie R_q$. When the multiway join has a property called *acyclicity* there exists an efficient oblivious algorithm for evaluating the join.

The algorithm for evaluating a multiway join is a generalization of the algorithm for binary join. Informally, we compute the contribution of each tuple in $R_1, \ldots, R_q$ towards the final join. The

Id	*A*	*N*	N_S
1	*a*	1	3
2	*b*	1	1
3	*a*	1	3

(a): $\tilde{R}$

Id	*A*	*N*	N_R	*JId*
1	*a*	1	2	1
2	*b*	1	1	1
3	*a*	1	2	2
4	*a*	1	2	3

(b): $\tilde{S}$

Id	*A*	*Jid*
1	*a*	1
3	*a*	1
1	*a*	2
3	*a*	2
1	*a*	3
3	*a*	3
2	*b*	1

(c): $\langle R_e \rangle_{A,Jid}$

Id	*A*	*Jid*
1	*a*	1
1	*a*	1
3	*a*	2
3	*a*	2
4	*a*	3
4	*a*	3
2	*b*	1

(d): $\langle S_e \rangle_{A,Jid}$

Figure 6: Intermediate tables used by Algorithm 4 for Example of Figure 2. Only relevant columns of R_{exp} and S_{exp} are shown.

contribution generalizes the notion of a join-graph degree in the binary join case, and this quantity can be computed by performing a sequence of semi-join aggregations between the input relations. We expand the input relations $R_1, \ldots, R_q$ to $R_{1,exp}, \ldots, R_{q,exp}$ respectively by duplicating each tuple as many times as its contribution, and stitch the expanded tables to produce the final join output. The details of ordering the expansions $R_{1,exp}, \ldots, R_{q,exp}$ are now more involved. A formal description of the overall algorithm is deferred to the full-version [5]. Here we present a formal characterization of the class of multiway join queries our algorithm is able to handle.

DEFINITION 9. *The multiway join query $R_1 \bowtie \cdots \bowtie R_q$ is called* acyclic*, if we can arrange the relations $R_1, \ldots, R_q$ as nodes in a tree T such that for all $i, j, k \in [1, q]$ such that R_k is along the path from R_i to R_j in T, $Attr(R_i) \cap Attr(R_j) \subseteq Attr(R_k)$.*

THEOREM 10. *There exists an oblivious algorithm to compute the natural join query $(R_1 \bowtie \cdots \bowtie R_q)$ provided the query is acyclic. Further, the time complexity of the algorithm is $\Theta(n \log n + m \log m)$ where $n = \sum_i |R_i|$ is the input size and $m = |R_1 \bowtie \cdots \bowtie R_q|$ denotes the output size, and the TM memory requirement is $O(\log(n + m))$.*

The concept of acyclicity is well-known in database theory [34]; in fact, it represents the class of multiway join queries for which algorithms polynomial in input and output size are known. We use the acyclicity property to compute the contribution of each tuple towards to the final output using a series of semi-join aggregations. Without the acyclicity property we do not know of a way of computing this quantity short of evaluating the full join.

5.3 Grouping and Aggregation

This section presents an oblivious algorithm for grouping aggregation over acyclic joins. We present our algorithm for the case of SUM; it can be easily adapted for the other standard aggregation functions: MIN, MAX, AVG, and COUNT. The algorithm handles only a limited form of grouping where all the grouping attributes belong to a single relation. The query is therefore of the form $\mathbb{G}_{\mathcal{G}}^{\mathrm{SUM}(R_a.X)}$ $(R_1 \bowtie \cdots \bowtie R_q)$, where (wlog) $\mathcal{G} \in Attr(R_1)$. We believe the case where the grouping attributes come from multiple relations is hard as we discuss in Section 6.

Algorithm 5 Grouping and Aggregation: $\mathbb{G}_{\mathcal{G}}^{\mathrm{SUM}(R_a.X)}(R_1 \bowtie \cdots \bowtie R_q)$

```
 1: procedure GroupingAggr($(R_1, \ldots, R_q), \mathcal{G}, R_a.X$)
 2:     for $i = q$ to $1$ do
 3:         $\tilde{R}_i \leftarrow R_i$
 4:         if $\#c(i) = 0$ then $\tilde{R}_i \leftarrow \tilde{R}_i.(N \leftarrow 1)$                ▷ leaf table
 5:         else
 6:             for $j = 1$ to $\#c(i)$ do
 7:                 $\tilde{R}_i \leftarrow \tilde{R}_i.(N_{c(i,j)} \overset{\ltimes}{\leftarrow} Sum(\tilde{R}_{c(i,j)}.N))$
 8:             endfor
 9:             $\tilde{R}_i \leftarrow \tilde{R}_i.(N \leftarrow \Pi_{j=1}^{\#c(i)} N_{c(i,j)})$
10:         end if
11:         if $R_i = R_a$ then
12:             $\tilde{R}_a \leftarrow \tilde{R}_a.(S_X \leftarrow X \times N)$
13:         else if $R_a \in \mathrm{Desc}(R_i)$ then
14:             $\ell \leftarrow$ unique $\ell$ such that $R_a \in \mathrm{Desc}(R_{c(i,\ell)})$
15:             $\tilde{R}_i \leftarrow \tilde{R}_i.(S_X \overset{\ltimes}{\leftarrow} Sum(R_{c(i,\ell)}.S_X))$
16:         end if
17:     endfor
18:     $\tilde{R}_1 \leftarrow \tilde{R}_1.(Id_{\mathcal{G}} \leftarrow \mathrm{ID}_{\mathcal{G}})$
19:     $\tilde{R}_1 \leftarrow \tilde{R}_1.(RS_X \leftarrow \mathrm{RSum}_{\mathcal{G}}^{-Id_{\mathcal{G}}})$
20:     Output $\pi_{\mathcal{G},RS_X}(\sigma_{Id_{\mathcal{G}}=1}(\tilde{R}_1))$
21: end procedure
```

Notation: Since the join $R_1 \bowtie \cdots \bowtie R_q$ is acyclic we can arrange the relations as nodes in a tree T as per Definition 9. An algorithm for constructing such a tree is presented in [35]. For the remainder of this section fix some tree T. Wlog, we assume that relations $R_1, \ldots, R_q$ are numbered by a pre-order traversal of tree T so that if R_i is an ancestor of R_j then $i < j$. For any relation R_i, we use $\#c(i)$ to denote the number of children and $R_{c(i,1)}, \ldots, R_{c(i,\#c(i))}$ to denote the children of R_i in T; we denote the parent of R_i using $R_{p(i)}$. We use $\mathrm{Desc}(R_i)$ and $\mathrm{Anc}(R_i)$ to denote the descendants and ancestors of R_i in T; both $\mathrm{Anc}(R_i)$ and $\mathrm{Desc}(R_i)$ contain R_i. For any set $\mathcal{R}$ of relations $\bowtie \mathcal{R}$ denotes the natural join of elements of $\mathcal{R}$; e.g., $\bowtie \mathrm{Desc}(R_i)$ denotes the join of R_i and it descendants.

Algorithm 5 presents our grouping aggregation algorithm. The algorithm operates in 2 stages: (1) a bottom-up counting stage and (2) a grouping stage which works over just R_1.

Bottom-up Counting: In this stage, we add an attribute N to each tuple. For a tuple $t \in R_i$, $t[N]$ denotes the number of join tuples t is part of in $\bowtie \mathrm{Desc}(R_i)$. For leaf relations R_i, $t[N] = 1$ for all tuples $t \in R_i$. For non-leaf relations, a simple recursion can be used to compute the value of attribute N. Consider $t \in R_i$ for some non-leaf R_i and let $t[N_{c(i,j)}]$ denote the number of join tuples t is part of in the join $\bowtie (\{R_i\} \cup \mathrm{Desc}(R_{c(i,j)})$ (join of all descendants rooted in child $R_{c(i,j)}$). Then we can show using the acyclic property of the join, $t[N] = \Pi_j t[N_{c(i,j)}]$ (Step 9). In addition, for all relations in $\mathrm{Anc}(R_a)$ (R_a is the relation containing aggregated column X), we add a partial aggregation attribute S_X. For a tuple $t \in R_i \in \mathrm{Anc}(R_a)$, $t[S_X]$ represents the sum of $R_a.X$ values in $\bowtie \mathrm{Desc}(R_i)$ considering only tuples that t is part of.

Grouping: This stage essentially performs the grouping $\mathbb{G}_{\mathcal{G}}^{\mathrm{SUM}(S_X)}(R_1)$. We attach a unique id $Id_{\mathcal{G}}$ to each tuple within a group defined by $\mathcal{G}$ (Step 18). We then compute the running sum of S_X within each group defined by $\mathcal{G}$; we compute the running sum in descending order of $Id_{\mathcal{G}}$ so that the total sum for a group is stored with the record with $Id_{\mathcal{G}} = 1$. We get the final output by (obliviously) selecting the records with $Id_{\mathcal{G}}$ and performing suitable projections (Step 20).

THEOREM 11. *Algorithm 5 obliviously computes the grouping aggregation* $\mathbb{G}_{\mathcal{G}}^{\mathrm{SUM}(R_a.X)}(R_1 \bowtie \cdots \bowtie R_q)$*,* $\mathcal{G} \in Attr(R_1)$*, where* $(R_1 \bowtie \cdots \bowtie R_q)$ *is acyclic, in time* $\Theta(n \log n)$ *and using* $O(\log n)$ *TM memory, where* $n = \sum_i |R_i|$ *denotes the total size of the input relations.*

5.4 Selections

This section discusses how selections in SPJ and GSPJ queries can be handled. We assume a selection predicate P is a conjunction of table-level predicates, i.e., of the form $(P_{R_{i1}} \wedge P_{R_{i2}} \wedge \cdots)$ where each $P_{R_{ij}} : \mathcal{D}(R_{i_j}) \rightarrow \{\text{true}, \text{false}\}$ is a binary predicate over tuples of R_{i_j}. To handle selections in a GSPJ query $\mathbb{G}_{\mathcal{G}}^{F(A)}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$, we modify the bottom-up counting stage of Algorithm 5 as follows: when processing any relation R_i with a table-level predicate P_{R_i} that is part of P, we set the value of attribute N to 0 for all tuples $t_i \in R_i$ for which $P_{R_i}(t_i) = \text{false}$; for all other tuples the value of attribute N is calculated as before. We can show that the resulting algorithm correctly and obliviously evaluates the query $\mathbb{G}_{\mathcal{G}}^{F(A)}(\sigma_P(R_1 \bowtie \cdots \bowtie R_q))$. A similar modification works for the SPJ query $\sigma_P(R_1 \bowtie \cdots \bowtie R_q)$ and is described in the full version of the paper [5].

5.5 Exploiting Foreign Key Constraints

We informally discuss how we can exploit key-foreign key constraints; We note that keys and foreign keys are part of database schema (metadata), and we view them as public knowledge (see discussion in Appendix A). Consider a query Q involving a multiway join $R_1 \bowtie \cdots \bowtie R_q$ (with or without grouping) and let R_i (key side) and R_j (foreign key side) denote two relations involved in a key-foreign key join. We explicitly evaluate $R_{ij} \leftarrow R_i \bowtie R_j$ using the oblivious binary join algorithm and replace references to R_i and R_j in Q with R_{ij}. From key-foreign key property, it follows that $|R_{ij}| = |R_j|$, so this step by itself does not render the query processing non-oblivious. We treat any foreign key references to R_j as references to R_{ij}. We continue to this process of identifying key-foreign key joins and evaluating them separately until no more such joins exist. At this point, we revert to the general algorithms presented in Sections 5.2-5.4 to process the remainder of the query.

THEOREM 12. *There exists an oblivious (secure) query processing algorithm that requires* $O(\log n)$ *storage in TM for any SPJ or GSPJ query involving joins, grouping aggregation (as the outermost operation), and filters, if (1) the non-foreign key join graph is acyclic and (2) all grouping attributes are connected through foreign key joins, where* n *denotes the sum of query input and output sizes. Further, assuming no auxiliary structures, the running time of the algorithm is within* $O(\log n)$ *(multiplicative factor) of the running time of the best insecure algorithm.*

PROOF. (Sketch) From Theorems 8-11 and the informal descriptions in Section 5.4 and 5.5, it follows that there exist oblivious query processing algorithms for above class of queries that run in $O(n \log n)$ time and require $O(\log n)$ TM memory. Further, any algorithm requires $\Omega(n)$ time without auxiliary structures. □

THEOREM 13. *For the class of queries in Theorem 12 there exists an oblivious (secure) algorithm with I/O complexity within multiplicative factor* $\log_{M/B}(n/B)$ *of that of the optimal insecure algorithm, where* B *is the block size and* M *is the TM memory.*

PROOF. (Sketch) All of our algorithms are simple scans except for the steps that perform oblivious sorting. The I/O complexity of oblivious sorting is $O(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ [13] from which the theorem follows. □

6. HARDNESS ARGUMENTS

Section 5 presented efficient oblivious algorithms for evaluating (G)SPJ queries when the underlying join was acyclic and all grouping columns belonged to a single relation. All of our algorithms have time complexity $O((n+m)\log(n+m))$ where n and m and input and output sizes, respectively. Any algorithm requires $\Omega(n+m)$ time without pre-processing, so our algorithms are within a $\log(n+m)$ factor away from an instance optimal algorithm. In the following, we call such oblivious algorithms *instance efficient*. While we do not have formal proofs, we present some evidence that suggests that instance efficient oblivious algorithms for cyclic joins and for the case where grouping columns come from different tables seem unlikely to exist.

At a high level, our arguments rely on the following intuition: if a query Q does not have a near-linear algorithm (with time complexity $(n+m)\text{polylog}(n+m)$) in the worst case it is unlikely to have an oblivious algorithm since its behavior on an easy instance would be different from that on a worst case instance. There are some difficulties directly formalizing this intuition since some of the computation occurs within TM potentially without an externally visible data access.

Recent work has identified the following *3SUM* problem as a simple and useful problem for polynomial time lower-bound reductions [26]: Given an input set of n numbers identify $x, y, z \in S$ such that $x + y = z$. There exists a simple $O(n^2)$ algorithm for 3SUM: Store the n numbers in S in a hashtable $\mathcal{H}$. Consider all pairs of numbers $x, y \in S$ and check if $x + y \in \mathcal{H}$. It is widely believed that this algorithm is the best possible and [26] uses this 3SUM-hardness conjecture to establish lower bounds for a variety of combinatorial problems.

We introduce the following simple variant of 3SUM-hardness that captures the additional complexity of TM computations. In this variant, an algorithm has access to a *cache* of size n^δ ($\delta < 1$) words. Access to the cache is free while accesses to non-cache memory have a unit (time) cost. We conjecture that having access to a free cache does not bring down the asymptotic complexity of 3SUM. For example, in the hashtable based solution above, only a small part of the input (at most n^δ) can be stored in the cache and most of the hashtable lookups ($n - n^\delta$) incur a non-cache access cost.

CONJECTURE 14. *(3SUM-Cache(δ)-hardness) Any algorithm for 3SUM with input size* n *having access to a free cache of size* n^δ *requires* $\Omega(n^{2-o(1)})$ *time in expectation.*

Assuming this conjecture is true, the algorithms of Section 5 almost represent a characterization of the class of single-block queries that have an instance efficient oblivious algorithm.

THEOREM 15. *There does not exist an instance efficient oblivious algorithm with a TM with memory* n^δ *for cyclic joins unless there exists a subquadratic* $O(n^{2-\Omega(1)})$ *algorithm for 3SUM-Cache(δ), where* n *denotes the sum of input and output sizes of the join.*

PROOF. Enumerating m triangles in a graph with m edges in time $O(m^{4/3-\epsilon})$ is 3SUM-hard [26] (Theorem 5). Enumerating triangles can be expressed as a cyclic join query over the edge relation. It follows that evaluating a cyclic join query in $O(n^{4/3-\epsilon})$ is 3SUM-hard. The free cache does not affect this reduction from 3SUM to cyclic join evaluation, so evaluating a cyclic join query using a TM with cache n^δ with $O(n^{4/3-\epsilon})$ UM memory accesses is 3SUM-Cache(δ)-hard. We can construct easy instances for all sufficiently large n that only require $O(n)$ processing time. It follows that there does not exist an instance efficient oblivious algorithm for cyclic joins. □

The *set intersection enumeration* problem is the following: given k sets $S_1, \ldots, S_k$, $S_i \subseteq \mathcal{U}$ drawn from some universe $\mathcal{U}$, identify all pairs (i, j) such that $S_i \cap S_j \neq \phi$. A simple algorithm is to build an inverted index that stores for each element $e \in S_1 \cup \ldots \cup S_k$ the list of integers j such that $e \in S_j$. We consider all pairs of integers within each list and output the pair if we have not already done so. This simple algorithm is quadratic in the input and output sizes in the worst case. The set intersection enumeration problem is fairly well-studied and is at the core of most high-dimensional [15] and approximate string matching [3] but no asymptotically better algorithm is known. There is a simple reduction from set intersection enumeration to evaluating grouping queries where grouping attributes come from multiple relations.

THEOREM 16. *If there exists a $O((n + m)\text{polylog}(n + m))$ time algorithm for evaluating $\mathbb{G}_{A,B}(R(A, Id) \bowtie S(B, Id))$ where n and m are input and output sizes of the query, then there exists an $O((n+m)\text{polylog}(n+m))$ time algorithm for set intersection enumeration.*

PROOF. We encode the input to set intersection enumeration as two relations $R(A, Id)$ and $S(B, Id)$. The domain of A and B is $\mathcal{U}$ the universe of elements. For each $e \in S_i$ we include a tuple (e, i) in both R and S. (R and S are therefore identical.) The theorem follows from the observation that the desired output of set intersection enumeration is precisely $\mathbb{G}_{A,B}(R(A, Id) \bowtie S(B, Id))$. □

Theorem 16 along with the fact that we can construct an input instance for the query $\mathbb{G}_{A,B}(R(A, Id) \bowtie S(B, Id))$ which can be evaluated in $O((n + m)\text{polylog}(n + m))$ time suggests that an oblivious algorithm for this query is likely to imply a (significantly) better algorithm for set intersection enumeration than currently known.

7. RELATED WORK

While we focused mostly on systems that use trusted hardware for querying encrypted data, there exist other approaches. One such approach relies on *homomorphic encryption* that allows computation directly over encrypted data; e.g., the *Paillier cryptosystem* [25] allows us to compute the encryption of $(v_1 + v_2)$ given the encryptions of v_1 and v_2 without requiring the (private) encryption key, and can be used to process SUM aggregation queries [9]. However, despite recent advances, practical homomorphic encryption schemes are currently known only for limited classes of computation. There exist simple queries that the state-of-the-art systems [27] that rely solely on homomorphic encryption cannot process. A second approach is to use the client as the trusted location inaccessible to the adversary [14, 32]. One drawback of using the client is that some queries might necessitate moving large amounts of data to the client for query processing and defeat the very purpose of using a cloud service. We can reduce some of the above drawbacks by combining homomorphic encryption with the client processing approach [32], but, given the current state-of-the-art of homomorphic encryption, a comprehensive and robust solution to querying encrypted data seems to require the trusted hardware architecture we have assumed in this paper.

8. ACKNOWLEDGMENTS

Discussions with the Cipherbase team greatly helped us at all stages of this work. We thank Chris Re, Atri Rudra, and Hung Ngo for pointing us to hardness of enumerating triangles result from [26], and Bryan Parno and Paraschos Koutris for valuable feedback on initial drafts. We also thank Dan Suciu and anonymous reviewers for helping better position our work against ORAM simulation.

9. REFERENCES

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[2] Amazon Corporation. Amazon Relational Database Service. http://aws.amazon.com/rds/.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] Arvind Arasu, Spyros Blanas, Ken Eguro, et al. Orthogonal security with cipherbase. In *CIDR*, 2013.

[5] Arvind Arasu and Raghav Kaushik. Oblivious query processing. *CoRR*, abs/1312.4012, 2013.

[6] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD Conference*, pages 205–216, 2011.

[7] Cipherbase project. http://research.microsoft.com/en-us/projects/dbencryption/.

[8] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, et al. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[9] Tingjian Ge and Stanley B. Zdonik. Answering aggregation queries in a secure system model. In *VLDB*, pages 519–530, 2007.

[10] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[11] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27, 2011.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, et al. Practical oblivious storage. In *CODASPY*, pages 13–24, 2012.

[13] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011.

[14] H. Hacigümüs, B. R. Iyer, C. Li, et al. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD Conference*, 2002.

[15] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[16] IBM Corporation. IBM Systems cryptographic hardware products. http://www-03.ibm.com/security/cryptocards/.

[17] Eddie Kohler. Hot crap! In *WOWCS*, 2008.

[18] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[19] Yaping Li and Minghua Chen. Privacy preserving joins. In *ICDE*, pages 1352–1354, 2008.

[20] Martin Maas, Eric Love, Emil Stefanov, et al. PHANTOM: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[21] Microsoft Corporation. SQL Azure. http://www.windowsazure.com/en-us/home/features/sql-azure/.

[22] Microsoft Corporation. SQL Server Encryption. http://technet.microsoft.com/en-us/library/bb510663.aspx.

[23] Oracle Corporation. Transparent Data Encryption. http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html.

[24] Anna Pagh and Rasmus Pagh. Scalable computation of acyclic joins. In *PODS*, pages 225–232, 2006.

[25] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[26] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610, 2010.

[27] R. A. Popa, C. M. S. Redfield, N. Zeldovich, et al. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[28] An SME perspective on cloud computing (survey). European Network and Information Security Agency (ENISA), 2009.

[29] Emil Stefanov, Marten Van Dijk, Elaine Shi, et al. Path ORAM: An extremely simple oblivous RAM protocol. In *CCS*, 2013.

[30] Germany tackles tax evasion. Wall Street Journal, Feb 7 2010.

[31] The TPC-H Benchmark. `http://www.tpc.org`.

[32] Stephen Tu, M. Frans Kaashoek, Samuel Madden, et al. Processing analytical queries over encrypted data. In *VLDB*, 2013.

[33] P. Williams and R. Sion. Usable pir. In *NDSS*, 2008.

[34] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

[35] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *Comp. Soft. and Appln. Conf*, pages 306–312, 1979.

APPENDIX

A. QUERY AND METADATA SECURITY

Ideally a secure query processing system hides both database contents and queries being evaluated against the database. Given how databases are typically used, we believe it is important to study the security of database contents even if the adversary has full knowledge of the query. Databases are typically accessed using a front end application, and the entropy of the queries run over the database is typically small given knowledge of the application. As a concrete example, consider a paper review system like EasyChair, which is a web application that presumably stores and queries papers and review information using a backend database. The web application might itself be well known (or open source [17]), so the kinds of queries issued to the database is public knowledge.

That said, the following question remains: To what extent should a secure query processing system hide queries? What should an adversary without knowledge of the query be able to learn? We argue that perfect query security—being able to mask whether a query is a single table filter or a 50-table join—is impractical given the richness of database query languages. Going the extra mile to hide queries is also wasteful if we assume knowledge of the application is easy to acquire as in the discussion of the paper review system above.

What an application (source code) does not usually specify is actual query constants which might be generated, e.g., by users filling in forms or from values in the database and it might be valuable (and as it turns out, easy) to secure these. Accordingly, in this paper we equate query security with *query constants* security: an adversary might learn by observing a query execution, that e.g., the query is a 2-way join with a filter on the first table, but she does not learn the filter predicate constants. We note that this is either explicitly [14, 27, 32] or implicitly [4, 6] the notion of query security used in prior work.

A related issue is metadata security. Metadata refers to information such as number of tables and the schema (column names and types) of each table. While column and table names can be easily anonymized, for the same reasons mentioned in query security, formally securing metadata is both difficult and not very useful if the adversary has knowledge of the application.