OBSCURE: Information-Theoretic Oblivious and Verifiable Aggregation Queries

Yin Li[1], Sharad Mehrotra[2], Nisha Panwar[2], Shantanu Sharma[2], and Sumaya Almanee[2]
[1]Xinyang Normal University, China. [2]University of California, Irvine, USA.

***Abstract*—Despite extensive research on cryptography, secure and efficient query processing over outsourced data remains an open challenge. This paper develops communication-efficient and information-theoretic secure algorithms for privacy-preserving aggregation queries using multi-party computation (MPC). More specifically, the paper builds query processing techniques over secret-shared data outsourced by single or multiple database owners. These algorithms allow a user to execute the queries without involving the database owner and also prevent the network and the (adversarial) clouds to learn the user's queries, results, or the database. We further provide (non-mandatory) privacy-preserving result verification algorithms that detect some malicious behaviors, which cannot be detected by a Byzantine tolerant protocol. Experimental evaluations show that the proposed algorithm outperforms an industrial MPC-based system.**

I. INTRODUCTION

Database-as-a-service (DaS) model [1] allows authenticated users to execute their queries on an untrusted public cloud. Over the last two decades, several cryptographic techniques (*e.g.*, [2]–[6]) have been proposed to achieve secure and privacy-preserving computations in the DaS model. These techniques can be broadly classified based on the cryptographic security into two categories, as follows:

Computational-secure techniques that assume the adversary lacks adequate computational capabilities to break the underlying cryptographic mechanism in polynomial time. Non-deterministic encryption [2], homomorphic encryption [4], order-preserving encryption (OPE) [5], and searchable-encryption [3] are examples of such techniques. Homomorphic encryption mixed with oblivious-RAM (ORAM) offers the most computational-secure mechanisms.

Information-theoretic secure techniques that are unconditionally secure and independent of adversary's computational capabilities. Shamir's secret-sharing (SSS) [6] is a well-known information-theoretic secure protocol. In SSS, multiple (secure) shares of a dataset are kept at different clouds that do not communicate with each other, such that a single cloud cannot learn anything about the data. Secret-sharing-based techniques are secure under the assumption that a majority of the clouds (equal to the threshold of the secret-sharing mechanism) do not collude. Secret-sharing mechanisms also have applications in other areas such as Byzantine agreement, secure multiparty computations (MPC), and threshold cryptography, as mentioned in [7].

The computational or information-theoretic secure database techniques may also be broadly classified into two categories, based on the supported queries, as follows: (*i*) *Techniques that support selection/join.* Different cryptographic techniques are built for selection queries; for example, searchable encryption, deterministic/non-deterministic encryption, OPE, and partially homomorphic techniques; and (*ii*) *Techniques that support aggregation.* Cryptographic techniques that exploit homomorphic mechanisms are suitable for this category; for example, homomorphic encryption, SSS, or MPC techniques.

While much of secure data outsourcing have been built around computationally secure cryptographic mechanisms, recent works, both in academia and industries/startups have begun to explore information-theoretically secure techniques, due to higher levels of security offered,[1] as well as, relatively efficient support data analytics and OLAP operations involving aggregate functions (*e.g.*, count, sum, maximum, and minimum). For instance, products/industries/startups, such as Jana [9] by Galois, Pulsar [10] by Stealth Software, Sharemind [11] by Cybernetica, Unbound Tech., Partisia, Secret Double Octopus, SecretSkyDB Ltd., are MPC-based databases systems that offer strong security guarantees.

While MPC-based solutions (Jana, Pulsar, and Sharemind) require multiple clouds to collaborate to answer queries, secret-sharing-based mechanisms,[2] as in [12]–[14], do not require inter-cloud communication to answer queries. However, work on exploiting secret-sharing for data outsourcing suffers from several drawbacks. In particular, [12] uses a trusted-third-party to perform aggregation queries, based on an insecure order-preserving secret-sharing (OP-SS) mechanism to answer maximum/minimum queries. [13] requires the database (DB) owner to retain each polynomial, which was used to create shares of the database. Thus, the DB owner stores $n \times m$ polynomials, where n and m are the numbers of tuples and attributes in a relation. Also, [13] reveals access-patterns (*i.e.*, the identity of tuples that satisfy a query) and uses an insecure OP-SS. [14] used a novel string-matching operation over the shares at the cloud, but it cannot perform general aggregations with selection over complex predicates. In short, all the SSS-based works for aggregation queries either overburden the DB owner, are insecure due to OP-SS, reveal access-patterns, or support a very limited form of aggregation queries without any selection criteria.

We develop SSS-based algorithms (named OBSCURE) that support a large class of *access-pattern-hiding aggregation queries with selection*. OBSCURE supports count, sum, average, maximum, minimum, top-k, and reverse top-k, queries,

[1]Some of the computationally-secure mechanisms are vulnerable to computationally sufficiently powerful adversaries. For instance, Google, with sufficient computational capabilities, broke SHA-1 [8].

[2]See [7] for survey of variants of secret-sharing mechanisms.

while reveals nothing about data/query/results to an adversary. Also, we provide an oblivious result verification algorithm for aggregation queries, and ensure that an adversary does not learn anything about the verification. OBSCURE's verification step is not mandatory and depends on the querier's choice. Note that [15] and [16] provided a verification approach on secret-shared data. [15] considered verification process for MPC using a trusted-third-verifier. [16] provided operation verification (*i.e.*, whether all the desired tuples are scanned or not) for only sum queries, unlike OBSCURE that verifies results for all queries. Also, [16] overburdens the DB owner by keeping metadata for each tuple, which serves as a mechanism to sum verification. OBSCURE verification methods neither involve the DB owner to verify the results nor require a trusted-third-verifier.

Our proposed algorithms overcome several limitations/disadvantages of existing information-theoretic aggregate queries and verification solutions. First, we do not require the DB owner or a trusted party for answering a query or verifying the query results, unlike [12], [13], [15], [16], and hence, an authenticated querier can directly execute queries and verify the result of the queries. Second, we do not retain any tuple ids to be included in answering a query, unlike [16], and hence, we eliminate the role of the DB owner to answer a query. Third, as mentioned previously, OP-SS is not secure, especially, prone to background knowledge attacks, we use OP-SS for answering maximum and minimum, while also preventing background knowledge attacks on OP-SS by partitioning the data, unlike [12], [13], [17]. Note that Prio [18] supports a mechanism for confirming the maximum number, if the maximum number is known; however, Prio [18] does not provide any mechanism to compute the maximum/minimum. Fourth, in our scheme, the single cloud can never learn the database/query/outputs and which tuples of the database are included in answering a query, *i.e.*, *hiding access-patterns*, unlike [12], [13], [16], [17].

Fifth, our algorithms use minimal communication rounds between the user and each cloud, unlike Sepia [19] and [20] that use many rounds for doing basic *addition and less than* operations. Specifically, the count, sum, average, and their verification algorithms require only one communication round between each cloud and the user. However, maximum/minimum finding algorithms require at most four communication rounds. In addition, our scheme achieves the minimum communication cost for aggregate queries, especially for count, sum, and average queries, by aggregating data locally at each cloud. Finally, our algorithm supports aggregation queries[3] on a dataset outsourced by a single DB owner (*e.g.*, patient database outsourced by a single hospital) or multiple DB owners (*e.g.*, data from smart meters). In addition, it is important to mention that for string-matching operations on shares, we do not develop any new algorithm. We use the string-matching algorithm given in [21] that performs search operations on the share, but only supports single-dimensional count queries [14], [22].

[3]While [9], [12]–[14] have also tried to develop mechanisms to support selections and joins operations over the secret-shared data, all such methods [9], [12]–[14] leak information due to access-patterns or inefficient due to quadratic nature of underlying algorithms and transmit the entire dataset to the user, when executing selection or join operations. We do not focus on join or pure selection queries.

Contributions and outline. The primary contributions of this paper are as follows:

1) Non-interactive and SSS-based algorithms for answering conjunctive/disjunctive count and sum queries (§IV, §V). These algorithms work for the scenarios where either a single or multiple DB owners outsource their data to the clouds.
2) Non-interactive and SSS-based result verification algorithms for count (§IV-A) and sum (§V-A) queries.
3) Non-interactive and SSS-based algorithms for fetching tuples having maximum values in some attributes (§VI).
4) Non-interactive and SSS-based algorithms for knowing the maximum value over the tuples that are outsourced by multiple DB owners (§VIII-C), verifying the maximum value and a retrieved tuple (§VIII-A), and when fetching all tuples having the maximum values (§VIII-B).
5) Experimental evaluations against an industrial MPC-based system, and that show our algorithms outperform that system (§VII). Also, we analyze each algorithm with respect to information leakage at the cloud.
6) Non-interactive and SSS-based algorithms for finding the minimum, top-k, and reverse-top-k (§A) and group-by queries (§A).
7) Non-interactive and SSS-based algorithms for finding the location of the desired tuple in the database (§B).

II. Building Blocks of the Algorithms

This section provides an overview of Shamir's secret-sharing (SSS), string-matching operations over secret-shares, 2's complement-based signbit finding, and order-preserving secret-sharing (OP-SS).

Shamir's secret-sharing (SSS). SSS [6] is a cryptographic algorithm developed by Adi Shamir in 1979. In SSS, the DB owner divides a secret value, say S, into c different fragments, called *shares*, and sends each share to a set of c non-communicating participants/clouds. These clouds cannot know the secret S until they collect $c' < c$ shares, where c' is the threshold of SSS. In particular, the DB owner randomly selects a polynomial of degree $c'-1$ with $c'-1$ random coefficients, *i.e.*, $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{c'-1}x^{c'-1} \in \mathbb{F}_p[x]$, where $a_0 = S$, p is a prime number, and $\mathbb{F}_p$ is a finite field of order p. The DB owner distributes the secret S to the c clouds by placing $x = 1, 2, \ldots, c$ into $p(x)$. Based on any c' shares, one can use Lagrange interpolation [23] to construct the secret S. SSS allows an addition of shares, *i.e.*, if $s(a)_i$ and $s(b)_i$ are shares of two values a and b, respectively, at the cloud i, then the cloud i can compute an addition of a and b itself, *i.e.*, $a + b = s(a) + s(b)$, without knowing real values of a and b.

String-matching operation on secret-shares. Homomorphic Secret-Sharing (HSS) [24], [25] and Accumulating-Automata

(AA) [21] are two new string-matching techniques on secret-sharing. Here, we explain AA to show how string-matching can be done on secret-shares. Let D be the cleartext data. Let $S(D)_i$ ($1 \leq i \leq c$) be the i^{th} secret-share of D stored at the i^{th} cloud, and c be the number of *non-communicating* clouds. AA allows a user to search a pattern pt by creating c secret-shares of pt (denoted by $S(pt)_i$, $1 \leq i \leq c$), so that the i^{th} cloud can search the secret-shared pattern $S(pt)_i$ over $S(D)_i$. The result of the string-matching operation is either 1 of secret-share form, if $S(pt)_i$ matches with a secret-shared string in $S(D)_i$ or 0 of secret-share form, if it does not match.

Example. Suppose that the cleartext database includes only a single alphabet: a, which may be stored as a 26-bit vector, where only the first position is 1 and the others are 0. If we perform a *logical and* operation (that we refer to *bit-wise multiplication*) of this vector with any vector in the cleartext, then the resultant will be one, only when another vector has 1 at the first bit position and remaining bits are 0, *i.e.*, another vector corresponds to 'a.' Since this computation reveals the database as well as the search pattern, the DB owner can hide a by using the different polynomials for creating secret-shares of all these 26-bits. The user can also perform a similar operation on the search pattern. The resultant of bit-wise multiplication of the two vectors contains 26-bits of secret-share form. Note that the secret-shares of a string and a pattern are not identical. Hence, it prevents the adversary to know the relationship between the string and the pattern. Also, the comparison of a secret-shared string with a secret-shared pattern is not identical to the comparison of strings in the cleartext.

Note. In this paper, we use AA that has *unary representation*[4] as a building block, since secret-shares over unary representation is easier to explain. However, one can use any type of representation (*e.g.*, *binary*) when using AA or use a different private string-matching technique over secret-shares like HSS [24], [25].

2's complement-based sigbit computation. [26] provided 2's complement-based sigbit computation. We will use signbit to find if two numbers are equal or not, as follows: $A \geq B$ if $signbit(A - B) = 0$, and $A < B$ if $signbit(A - B) = 1$. Let $A = [a_n, a_{n-1}, \dots, a_1]$ be a n bit number and $B = [b_n, b_{n-1}, \dots, b_1]$ be a n bit number. 2's complement subtraction converts $B - A$ into $B + \bar{A} + 1$, where $\bar{A} + 1$ is 2's complement representation of $-A$. We start from the least significant bit (LSB) and go through the rest of the bits. The method inverts a_i (by doing $1 - a_i$, where $1 \leq i \leq n$), calculates $\bar{a}_0 + b_0 + 1$ and its carry bit. After finishing this on all the n bits, the most significant bit (MSB) keeps the signbit.

Order-preserving secret-sharing (OP-SS). The concept of OP-SS was introduced in [12] and maintains the order of the values in secret-shares too, *e.g.*, if v_1 and v_2 are two values in cleartext such that $v_1 < v_2$, then $S(v_1) < S(v_2)$ at any cloud. It is clear that finding records with maximum or minimum values using OP-SS are trivial. However, ordering revealed by OP-SS can leak more information about records. Consider, for instance, the following Table 1. For explanation purpose, we represent Table 1 in cleartext. In Table 1, the salary field can be stored using OP-SS. If we know (background knowledge) that employees in the *security* department make more money than others, we can infer from the representation that the second tuple corresponds to someone from the *security* department. Thus, OP-SS, by itself, offers little security. However, as we will see later in §VI, by splitting the fields such as *salary* that can be stored using OP-SS while storing other fields using SSS. Thus, we can benefit from the ordering supported by OP-SS, without compromising on security.

EmpID	Name	Salary	Dept
E101	John	1000	Testing
E101	John	100000	Security
E102	Adam	5000	Testing
E103	Eve	2000	Design
E104	Alice	1500	Design
E105	Mike	2000	Design

Figure 1: A relation: Employee.

III. Preliminary

This section provides a description of entities, the data outsourcing model, the adversarial model, and security properties for obliviously executing aggregation and verification queries.

A. The Model

We assume the following three entities in our model.

(*i*) A set of c *untrusted* and *non-communicating* clouds. The clouds do not exchange data with each other to compute any answer. The only possible data exchange of a cloud is with the user/querier or the database owner.

(*ii*) The *trusted* database (DB) owner, that creates c secret-shares of the data and transfers the i^{th} share to the i^{th} cloud. The secret-shares are created by an algorithm that supports *non-interactive* addition and multiplication of two shares, which is required to execute the private string-matching operation, at the cloud, as explained in §II.[5]

(*iii*) An (authenticated and authorized) user/querier, who executes queries on the secret-shared data at the clouds. The query must be sent to at least $c' < c$ number of clouds, where c' is the threshold of Shamir's secret-sharing. The user fetches the partial outputs from the clouds and performs a simple operation (polynomial interpolation using Lagrange polynomials [23]) to obtain the secret-value.

Data model. The DB owner wishes to outsource a relation R having attributes $A_1, A_2, \dots, A_m$ and n tuples, and creates the following two relations R^1 and R^2:

• *Relation* R^1 that consists of all the attributes $A_1, A_2, \dots, A_m$ along with two additional attributes, namely TID (tuple id) and Index. As will become clear, the TID attribute will help in finding tuples having the maximum/minimum/top-k values, and the Index attribute will be used to know the

[4]Our unary representation uses significantly fewer bits as compared to Prio's [18] unary representation.

[5]Note that the choice of the underlying non-interactive and string-matching-based secret-sharing mechanism does not change our proposed aggregation and verification algorithms.

EmpID	Name	Salary	Dept	TID	Index
E101	John	1000	Testing	3	3
E101	John	100000	Security	2	2
E102	Adam	5000	Testing	5	5
E103	Eve	2000	Design	4	4
E104	Alice	1500	Design	1	1
E105	Mike	2000	Design	6	6

(a) $R^1 =$ `Employee1` relation.

CTID	SSTID	Salary
1	1	1500
5	5	5000
3	3	1000
6	6	2000
2	2	100000
4	4	2000

(b) $R^2 =$ `Employee2` relation.

Figure 2: Two relations obtained from `Employee` relation.

tuples satisfying the query predicate. The i^{th} values of the `TID` and `Index` attributes have the *same and unique* random number between 1 to n.

• *Relation* R^2 that consists of three attributes `CTID` (cleartext tuple id), `SSTID` (secret-shared tuple id), and an attribute, say A_c, on which a comparison operator (minimum, maximum, and top-k) needs to be supported.[6] It will be clear in §VI why does the DB owner outsource two relations R^1 and R^2. The i^{th} values of the attributes `CTID` and `SSTID` of the relation R^2 keep the i^{th} value of the `TID` attribute of the relation R^1. The i^{th} value of the attributes A_c of the relation R^2 keeps the i^{th} value of an attribute of the relation R^1 on which the user wants to execute a comparison operator. Further, the tuples of the relations R^2 are permuted, and this permutation is kept secret. The reason of doing permutation is that the adversary cannot relate any tuple of both the secret-shared relations, which will be clear soon.

Example. Consider the `Employee` relation (see Figure 1). The DB owner creates R^1 = `Employee1` relation[7] (see Figure 2a) with `TID` and `Index` attributes. Further, the DB owner creates R^2 = `Employee2` relation (see Figure 2b) having three attributes `CTID`, `SSTID`, and `Salary`.

Creating secret-shares. Let $A_i[a_j]$ ($1 \leq i \leq m+1$ and $1 \leq j \leq n$) be the j^{th} value of the attribute A_i. The DB owner creates c secret-shares of each attribute value $A_i[a_j]$ of the relation R^1 using a secret-sharing mechanism that allows string-matching operations at the cloud (as specified in §II). However, c shares of the j^{th} value of the attribute A_{m+2} (*i.e.*, `Index`) are obtained using SSS. This will result in c relations: $S(R^1)_1$, $S(R^1)_2$, …, $S(R^1)_c$, each having $m+2$ attributes. The notation $S(R^1)_k$ denotes the k^{th} secret-shared relation of R^1 at the cloud k. We use the notation $A_i[S(a_j)]_k$ to indicate the j^{th} secret-shared value of the i^{th} attribute of a secret-shared relation at the cloud k.

Further, the DB owner creates c secret-shares of each value of the two attributes `SSTID` and A_c of the relation R^2, respectively, using a secret-sharing mechanism that allows string-matching operations on the clouds and using a insecure order-preserving secret-sharing [12], [13], [17]. The secret-shares of the relation R^2 are denoted by $S(R^2)_i$ ($1 \leq i \leq c$). The attribute `CTID` is outsourced in cleartext with the shared relation $S(R^2)_i$. It is important to mention that `CTID` attribute allows fast search due to cleartext representation than `SSTID` attribute, which allows search over shares.

Note that the DB owner's objective is to hide any relationship between the two relations when creating shares of the relations $S(R^1)$ and $S(R^2)$, *i.e.*, the adversary cannot know by just observing any two tuples of the two relations that whether these tuples share a common value in the attribute `TID`/`SSTID` and A_c or not. Thus, shares of an i^{th} ($1 \leq i \leq n$) value of the attribute `TID` in the relation $S(R^1)_j$ and in the attribute `SSTID` of the relation $S(R^2)_j$ must be different at the j^{th} cloud. In addition, by default, the attribute A_c have different shares in both the relations, due to the usage of different secret-sharing mechanisms for different attributes. Note that the representation above, even though it uses OP-SS does not suffer from attacks based on background knowledge, as mentioned in §II. The DB owner outsources the relations $S(R^1)_i$ and $S(R^2)_i$ to the i^{th} cloud.

B. Adversarial Model

We assume adversarial clouds that are *not trustworthy*, store secret-shared outsourced datasets, and (*i*) correctly compute assigned tasks without tampering and return answers; however, they may exploit side knowledge (*e.g.*, query execution, background knowledge, and the output size) to gain as much information as possible about the stored data; and (*ii*) deviate from the algorithm and delete any tuple of the relation.

The first type of adversarial behavior is identical to an honest-but-curious adversary, which is considered widely in many cryptographic algorithms and in the standard DaS query processing model, keyword searches, and aggregation queries [1], [27]–[29]. In literature, the second type of adversarial behavior is considered under a malicious adversary. Note that we do not assume any malicious user and malicious DB owners.

We assume that the adversary cannot know and collude all (or possibly the majority of) the clouds, and hence, the majority of communication channels between all the clouds and the user are unknown to the adversary. The adversary can eavesdrop on the minority of the communication channels between the cloud and the DB owner/user to gain knowledge about data, queries, or results. The channel is assumed to be secure, and only authenticated users can request a query on the clouds. The adversary is also aware of some public information, such as the actual number of tuples and number of attributes in a relation, which can be hidden by adding fake tuples and attributes.[8] Since we assume that the adversary cannot collude the majority of the clouds, which is more than the threshold of a secret-sharing mechanism, the adversary

[6]If there are x attributes on which comparison operators will be executed, then the DB owner will create x relations, each with attributes `CTID`, `SSTID`, and one of the x attributes.

[7]For verifying results of count and sum queries, we add two more attributes to this relation. However, we do not show here, since verification is not a mandatory step.

[8]The adversary cannot launch any attack against the DB owner. We do not consider cyber-attacks that can exfiltrate data from the DB owner directly, since defending against generic cyber-attacks is outside the scope of this paper.

cannot insert/update shares on the majority of the clouds.[9]

C. Security Properties

Our security definitions are identical to the standard security definition as in [31]–[33]. An algorithm is privacy-preserving if it maintains the privacy of the querier (*i.e.*, query privacy), the privacy of data from the clouds, and performs identical operations, regardless of the user query.

Query/Querier's privacy requires that the user's query must be hidden from the cloud, the DB owner, and the communication channel. In addition, the cloud cannot distinguish between two or more queries of the *same type* based on the output. Queries are of the same type based on their output size. For instance, all count queries are of the same type since they return almost an identical number of bits.

Definition: User's privacy. *For any probabilistic polynomial time adversarial cloud having a secret-shared relation $S(R)$ and any two input query predicates, say p_1 and p_2, the cloud cannot distinguish p_1 or p_2 based on the executed computations for either p_1 and p_2.*

Privacy from the cloud requires that the stored input data, intermediate data during a computation, and output data are not revealed to the cloud, and the secret value can only be reconstructed by the DB owner or an authorized user. In addition, two or more occurrences of a value in the relation must be different at the cloud to prevent frequency analysis while data at rest. Recall that due to secret-shared relations (by following the approach given in §III-A), the cloud cannot learn the relations and frequency-analysis, and in addition, due to maintaining the query privacy, the cloud cannot learn the query and the output.

Here, we must ensure that the cloud does not provide more information to the user, except for the response to the query (recall that user might be different compared to the data owner in our model). To show that we need to compare the real execution of the algorithm at the cloud against the ideal execution of the algorithm at a trusted party having the same data and the same query predicate. An algorithm maintains the data privacy from the cloud if the real and ideal executions of the algorithm return an identical answer to the user.

Definition: Privacy from the cloud. *For any given secret-shared relation $S(R)$ at a cloud, any query predicate qp, and any real user, say U, there exists a probabilistic polynomial time (PPT) user U' in the ideal execution, such that the outputs to U and U' for the query predicate qp on the relation $S(R)$ are identical.*

Properties of verification. We provide verification properties against malicious behaviors. A verification method must be oblivious and find any misbehavior of the clouds when computing a query. We follow the verification properties from [15], as follows: (*i*) the verification method cannot be refuted by the majority of the malicious clouds, and (*ii*) the verification method should not leak any additional information.

[9] Note that an adversary (who deviates from the algorithm and deletes any row) can be detected using Byzantine tolerant protocols [30], if the minority of the clouds misbehave.

Algorithms' performance. We analyze our oblivious aggregation algorithms on the following parameters, which are stated in Table I: (*i*) *Communication rounds*. The number of rounds required between the user and each cloud to obtain an answer to the query. (*ii*) *Computational cost at the cloud*. We measure the computational cost at the cloud in terms of the number of the rounds that the cloud performs to read the entire dataset. (*iii*) *Computational cost at the user*. The number of values/tuples that the user interpolates to know the final output.

Algorithms	Query conditions	Scan rounds at a cloud	Communication rounds	Interpolated values at user
Count §IV	SD	1	1	1
	CQ	1	1	1
	DQ	1	1	m
Sum §V	SD	1	1	1
	CQ	1	1	1
	DQ	1	1	m
Unconditional maximum/minimum (`SDBMax` §VI-A)	One occurrence with tuple	1	1	m
Conditional maximum/minimum (`SDBMax` §VI-B)	Finding maximum	1	2	$n+1$, $6\sqrt{n}+1$, or $\mathcal{T}+1$
	Tuple fetching	2	2	$n+m$, $6\sqrt{n}+m$, or $\mathcal{T}+m$
Maximum/Minimum (`MDBMax`) §VIII-C One /Multiple occurrences	Counting	$2n+1$	1	2
	Counting + tuple fetching	$2n+3$	3	$2\mathcal{T}+\ell m$
`SDBMax` many occurrences unconditional query §VIII-B	Finding	2	2	1
	Tuple fetching	2	2	$n+\ell m$
`SDBMax` many occurrences conditional query §VIII-B	Finding	2	2	1
	Tuple fetching	3	3	$2n+\ell m$
Group-by §A		1	1	g
Top-k or reverse top-k §A	Unique occurrence	1 or k	2 or 1	$k\times m$
Notations. m: # attributes. n: # tuples. D: the database $n\times m$. 1D: Single dimensional equality query. CE: Conjunctive equality query. DE: Disjunctive equality query. $\mathcal{T}$: # tuple ids satisfying a query predicate. ℓ: # tuples having the maximum/minimum in the desired attribute. g: # groups.				

Table I: Complexities of the algorithms.

D. OBSCURE *Overview*

Let us introduce OBSCURE at a high-level. OBSCURE allows single dimensional equality query, multi-dimensional conjunctive equality query, and multi-dimensional disjunctive equality query. Note that the method of OBSCURE for handling these types of queries is different from SQL, since OBSCURE does not support query optimization and indexing due to secret-shared data. Further, OBSCURE handles range-based queries by converting the range into equality queries. Executing a query on OBSCURE requires four phases, as follows:

PHASE 1: *Data upload by DB owner(s).* The DB owner uploads data to non-communicating clouds using a secret-sharing mechanism that allows addition and multiplication (*e.g.*, [21], [24], [25]) at the cloud, without involving the DB owner.

PHASE 2: *Query generation by the user.* The user generates a query, creates secret-shares of the query predicate, and sends them to the clouds. For generating secret-shares of the query predicate, the user follows the strategies given in §IV (count query), §V (sum queries), §VI (maximum/minimum), and §IV-A,§V-A (verification).

PHASE 3: *Query processing by the clouds.* The clouds process an input query in an oblivious manner such that they do not learn the query as well as the results satisfying the query. Finally, the clouds transfer their outputs to the user.
PHASE 4: *Result construction by the user.* The user performs Lagrange interpolation on the received results, which provide an answer to the query. The user can also verify these results by following the methods given in §IV-A, §V-A.

IV. COUNT QUERY AND VERIFICATION

In this section, we develop techniques to support count queries over secret-shared dataset outsourced by a single or multiple DB owners. The query execution does not involve the DB owner or the querier to answer the query. Further, we develop a method to verify the count query results.

Conjunctive count query. Our conjunctive equality-based count query scans the entire relation only once for checking single/multiple conditions of the query predicate. For example, consider the following conjunctive count query: `select count(*) from R where` $A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$.

The user transforms the query predicates to c secret-shares that result in the following query at the j^{th} cloud: `select count(*) from` $S(R)_j$ `where` $A_1 = S(v_1)_j \wedge A_2 = S(v_2)_j \wedge \ldots \wedge A_m = S(v_m)_j$. Note that the single dimensional query will have only one condition. Each cloud j performs the following operations:

$$Output = \sum_{k=1}^{k=n} \prod_{i=1}^{i=m} (A_i[S(a_k)]_j \otimes S(v_i)_j)$$

$\otimes$ shows a string-matching operation that depends on the underlying text representation. For example, if the text is represented as a unary vector, as explained in §II, $\otimes$ is a bit-wise multiplication whose results will be 0 or 1 of secret-share form. Each cloud j compares the query predicate value $S(v_i)$ against k^{th} value ($1 \leq k \leq n$) of the attribute A_i, multiplies all the resulting comparison for each of the attributes for the k^{th} tuple. This will result in a single value for the k^{th} tuple, and finally, the cloud adds all those values. Since secret-sharing *allows the addition of two shares*, the sum of all the n resultant shares provides the occurrences of tuples that satisfy the query predicate of secret-share form in the relation $S(R)$ at the j^{th} cloud. On receiving the values from the clouds, the user performs Lagrange interpolation [23] to get the final answer in the cleartext.

Correctness. The occurrence of a k^{th} tuple will only be included when the multiplication of m comparison results will be 1 of secret-share form. Having only a single 0 as a comparison resultant over an attribute of k^{th} tuple produce 0 of secret-share form, and hence, the k^{th} tuple will not be included. Thus, all the correct occurrences over all the tuples are included that satisfy the query's `where` clause.

Disjunctive count query. Our disjunctive count query also scans the entire relation only once for checking multiple conditions of the query predicate, like the conjunctive count query. Consider, for example, the following disjunctive count query: `select count(*) from R where` $A_1 = v_1 \vee A_2 = v_2 \vee \ldots \vee A_m = v_m$

The user transforms the query predicates to c secret-shares that results in the following query at the j^{th} cloud: `select count(*) from` $S(R)_j$ `where` $A_1 = S(v_1)_j \vee \ldots \vee A_m = S(v_m)_j$ The cloud j performs the following operation:

$$Result_i^k = A_i[S(a_k)]_j \otimes S(v_i)_j, 1 \leq i \leq m$$

$$Output = \sum_{k=1}^{k=n} (((Result_1^k \texttt{ OR } Result_2^k) \texttt{ OR } Result_3^k) \ldots \texttt{ OR } Result_m^k)$$

To capture the `OR` operation for each tuple k, the cloud generates m different results either 0 or 1 of secret-share form, denoted by $Result_i$ ($1 \leq i \leq m$), each of which corresponds to the comparison for one attribute. To compute the final result of the `OR` operation for each tuple k, one can perform binary-tree style computation. However, for simplicity, we used an iterative `OR` operation, as follows:

$$temp_1^k = Result_1^k + Result_2^k - Result_1^k \times Result_2^k$$

$$temp_2^k = temp_1^k + Result_3^k - temp_1^k \times Result_3^k$$

$$\ldots$$

$$Output^k = temp_{m-1}^k + Result_m^k - temp_{m-1}^k \times Result_m^k$$

After performing the same operation of each tuple, finally, the cloud adds all the resultant of the `OR` operation ($\sum_{k=1}^{k=n} Output^k$) and sends to the user. The user performs an interpolation on the received values that corresponds to the answer to the disjunctive count query.

Correctness. The disjunctive counting operation counts only those tuples that satisfy one of the query predicates. Thus, by performing `OR` operation over string-matching resultants for an i^{th} tuple results in 1 of secret-share form, if the tuple satisfied one of the query predicates. Thus, the sum of the `OR` operation resultant surely provides an answer to the query.

Information leakage discussion. The user sends query predicates of secret-share form, and the string-matching operation is executed on all the values of the desired attribute. Hence, access-patterns are hidden from the adversary, so that the cloud cannot distinguish any query predicate in the count queries. The output of any count query is of secret-share form and contains an identical number of bits. Thus, based on the output size, the adversary cannot know the exact count as well as differentiate two count queries. However, the adversary can know whether the count query is single dimensional, conjunctive or disjunctive count query.

Complexities. The count query requires only one communication round between the user and each cloud. Each cloud scans all the tuples of the relation $S(R^1)$ only once, regardless of the query conditions.

A. Verifying Count Query Results

In this section, we describe how results of count query can be verified. Note that we explain the algorithms only for a single dimensional query predicate. Conjunctive and disjunctive predicates can be handled in the same way.

Here, the objective is to verify that (*i*) all the tuples of the databases are checked against the count query predicates, and (*ii*) all the answers to the query predicate (0 or 1 of secret-share form) are included in the answer. In order to verify both the conditions, the cloud performs two functions, f_1 and f_2, as follows:

$$op_1 = f_1(x) = \sum_{i=1}^{i=n}(S(x_i) \otimes o_i)$$

$$op_2 = op_1 + f_2(y) = op_1 + \sum_{i=1}^{i=n} f_2(S(y_i) \otimes 1 - o_i)$$

i.e., the cloud performs the functions f_1 and f_2 on some n values, (it will be clear soon what are these n values). In the above equation o_i is the output of the string-matching operation carried on the i^{th} value of an attribute, say A_j, on which the user wants to execute the count query. The cloud sends the outputs of the function f_1, denoted by op_1, and the sum of the outputs of f_1 and f_2, denoted by op_2, to the user. The outputs op_1 and op_2 ensure the count result verification and that the cloud has checked each tuple, respectively. The verification method for a count query works as follows:

The DB owner. For enabling a count query result verification over any attribute, the DB owner adds two attributes, say A_x and A_y, having initialized with one, to the relation R^1. The values of the attributes A_x and A_y are also outsourced of SSS form (not unary representations) to the clouds.

Clouds. Each cloud k executes the count query, as mentioned in §IV, *i.e.*, it executes the private string-matching operation on the i^{th} ($1 \leq i \leq n$) value of the attribute A_j against the query predicate and adds all the resultant values. In addition, each cloud k executes the functions f_1 and f_2. The function f_1 (and f_2) multiplies the i^{th} value of the A_x (and A_y) attribute by the i^{th} string-matching resultant (and by the complement of the i^{th} string-matching resultant). The cloud k sends the following three things: (*i*) the sum of the string-matching operation over the attribute A_j, as a result, say $\langle result\rangle_k$, of the count query, (*ii*) the outputs of the function f_1: $\langle op_1\rangle_k$, and (*iii*) the sum of outputs of the function f_1 and f_2: $\langle op_2\rangle_k$, to the user.

User-side. The user interpolates the received three values from each cloud, which result in $Iresult$, Iop_1, and Iop_2. If the cloud followed the algorithm, the user will obtain: $Iresult = Iop_1$ and $Iop_2 = n$, where n is the number of tuples in the relation, and it is known to the user.

Example. We explain the above method using the following query on the `Employee` relation (refer to Figure 1): `select count(*) from Employee where Name = 'John'`. Figure 3 shows the result of the private string-matching, functions f_1 and f_2 at a cloud. Note that for the purpose of explanation, we use cleartext values; however, the cloud will perform all operations over secret-shares. For the first tuple, when the cloud checks the first value of `Name` attribute against the query predicate, the result of string-matching becomes 1 that is multiplied by the first value of the attribute A_x, and results in 1. The complement of the resultant is multiplied by the first value of the attribute A_y, and results in 0. All the other tuples are processed in the same way. Note that for this query, $result = op_1 = 2$ and $op_2 = 6$, if cloud performs each operation correctly.

`Name`	String-matching results	f_1	f_2
John	1	1	0
John	1	1	0
Adam	0	0	1
Eve	0	0	1
Alice	0	0	1
Mike	0	0	1
	2	2	4

Figure 3: An execution of the count query verification.

Correctness. Consider two cases: (*i*) all the clouds discard an entire identical tuple for processing, or (*ii*) all the clouds correctly process each value of the attribute A_j, op_1, and op_2; however, they do not add an identical resultant, o_i ($1 \leq i \leq n$), of the string-matching operation. In the first case, the user finds $Iresult = Iop_1$ to be true. However, the second condition ($Iop_2 = n$) will never be true, since discarding one tuple will result in $Iop_2 = n-1$. In the second case, the clouds will send the wrong $result$ by discarding an i^{th} count query resultant, and they will also discard the i^{th} value of the attribute A_x to lead to $Iresult = Iop_1$ at the user-side. Here, the user, however, finds the second condition $Iop_2 = n$ to be false.

Thus, the above verification method always correctly verifies the count query result. The only condition when the above method will not work, when all (or the majority of) the clouds collude, and they create shares of the string-matching operation without executing string-matching operations.

V. Sum and Average Queries

The sum and average queries are based on the search operation as mentioned above in the case of conjunctive/disjunctive count queries. In this section, we briefly present sum and average queries on a secret-shared database outsourced by single or multiple DB owners. Then, we develop a result verification approach for sum queries.

Conjunctive sum query. Consider the following query: `select sum`(A_ℓ) `from` R `where` $A_1 = v_1 \wedge A_2 = v_2 \wedge \ldots \wedge A_m = v_m$.

In the secret-sharing setting, the user transforms the above query into the following query at the j^{th} cloud: `select sum`(A_ℓ) `from` $S(R)_j$ `where` $A_1 = S(v_1)_j \wedge A_2 = S(v_2)_j \wedge \ldots \wedge A_m = S(v_m)_j$. This query will be executed in a similar manner as conjunctive count query except for the difference that the i^{th} resultant of matching the query predicate is multiplied by the i^{th} values of the attribute A_ℓ. The j^{th} cloud performs the following operation on each attribute on which the user wants to compute the sum, *i.e.*, A_ℓ and A_q:

$$\sum_{k=1}^{k=n} A_\ell[S(a_k)]_j \times (\prod_{i=1}^{i=m}(A_i[S(a_k)]_j \otimes S(v_i)_j))$$

Correctness. The correctness of conjunctive sum queries is similar to the argument for correctness of conjunctive count queries.

Disjunctive sum query. Consider the following query: `select sum`(A_ℓ) `from` R `where` $A_1 = v_1 \vee A_2 = v_2 \vee \ldots \vee A_m = v_m$. The user transforms the query predicates to c secret-shares that results in the following query at the j^{th} cloud:

$$\texttt{select sum}(A_\ell)\ \texttt{from}\ S(R)_j$$
$$\texttt{where}\ A_1 = S(v_1)_j\ \vee\ A_2 = S(v_2)_j\ \vee\ \dots\ \vee\ A_m = S(v_m)_j$$

The cloud j executes the following computation:

$$Result_i^k = A_i[S(a_k)]_j \otimes S(v_i)_j, 1 \leq i \leq m, 1 \leq k \leq n$$
$$Output = \sum_{k=1}^{k=n} A_\ell[S(a_k)]_j \times (((Result_1^k\ \texttt{OR}\ Result_2^k)\ \texttt{OR}\ Result_3^k) \dots\ \texttt{OR}\ Result_n^k)$$

The cloud multiplies the k^{th} comparison resultant by the k^{th} value of the attribute, on which the user wants to execute the sum operation (*e.g.*, A_ℓ), and then, adds all the values of the attribute A_ℓ.

Correctness. The correctness of a disjunctive sum query is similar to the correctness of a disjunctive count query.

Average queries. In our settings, computing the average query is a combination of the counting and the sum queries. The user requests the cloud to send the count and the sum of the desired values, and the user computes the average at their end.

Information leakage discussion. The sum query works identically to the count query. The sum query, like the count query, hides the facts which tuples are included in the sum operation, and the sum of the values.

A. Result Verification of Sum Queries

Now, we develop a result verification approach for a single dimensional sum query. The approach can be extended for conjunctive and disjunctive sum queries. Let A_ℓ be an attribute whose values will be included by the following sum query. $\texttt{select sum}(A_\ell)\ \texttt{from}\ R\ \texttt{where}\ A_q = v$.

Here, the objective is to verify that (*i*) all the tuples of the databases are checked against the sum query predicates, $A_q = v$, and (*ii*) only all the qualified values of the attribute A_ℓ are included as an answer to the sum query. The verification of a sum query first verifies the occurrences of the tuples that qualify the query predicate, using the mechanism for count query verification (§IV-A). Further, the cloud computes two functions, f_1 and f_2, to verify both the conditions of sum-query verification in an oblivious manner, as follows:

$$op_1 = f_1(x) = \sum_{i=1}^{i=n} o_i(x_i + a_i + o_i)$$
$$op_2 = f_1(x) = \sum_{i=1}^{i=n} o_i(y_i + a_i + o_i)$$

i.e., the cloud performs the functions f_1 and f_2 on some n values, (it will be clear soon what are these n values). In the above equations, o_i is the output of the string-matching operation carried on the i^{th} value of the attribute A_q, and a_i be the i^{th} $(1 \leq i \leq n)$ value of the attribute A_ℓ. The cloud sends the sum of the outputs of the function f_1, denoted by op_1, and the outputs of f_2, denoted by op_2, to the user. Particularly, the verification method for a sum query works as follows:

The DB owner. Analogous with the count verification method, if the data owner wants to provide verification for sum queries, new attributes should be added. Thus, the DB owner adds two attributes, say A_x and A_y, to the relation R^1. The i^{th} values of the attributes A_x and A_y are any two arbitrary random numbers whose difference equals to $-a_i$, where a_i is the i^{th} value of the attribute A_ℓ. The values of the attributes A_x and A_y are also secret-shared using SSS. For example, in Figure 4, boldface numbers show these random numbers of the attribute A_x and A_y in cleartext.

Clouds. The clouds execute the above-mentioned sum query, *i.e.*, each cloud k executes the private string-matching operation on the i^{th} $(1 \leq i \leq n)$ value of the attribute A_q against the query predicate v and multiplies the resultant value by the i^{th} value of the attribute A_ℓ. The cloud k adds all the resultant values of the attributes A_ℓ.

Verify stage. The cloud k executes the functions f_1 and f_2 on each value x_i and y_i of the attributes A_x and A_y, by following the above-mentioned equations. Finally, the cloud k sends the following three things to the user: (*i*) the sum of the resultant values of the attributes A_ℓ, say $\langle sum_\ell \rangle_k$, (*ii*) the sum of the outputs of the string-matching operation carried on the attribute A_q, say $\langle sum_q \rangle_k$,[10] against the query predicate, and (*iii*) the sum of outputs of the functions f_1 and f_2, say $\langle sum_{f1f2} \rangle_k$.

User-side. The user interpolates the received three values from each cloud, which results in $Isum_\ell$, $Isum_q$, and $Isum_{f1f2}$. The user checks the value of $Isum_{f1f2} - 2 \times Isum_q$ and $Isum_\ell$, and if it finds equal, the cloud has correctly executed the sum query.

Example. We explain the above method using the following query on the `Employee` relation (refer to Figure 1): `select sum(*) from Employee where Dept = 'Testing'`. Figure 4 shows the result of the private string-matching (o), the values of the attributes A_x and A_y in boldface, and the execution of the functions f_1 and f_2 at a cloud. Note that for the purpose of explanation, we show the verification operation in cleartext; however, the cloud will perform all operations over secret-shares.

For the first tuple, when the cloud checks the first value of `Dept` attribute against the query predicate, the string-matching resultant, o_1, becomes 1 that is multiplied by the first value of the attribute `Salary`. Also, the cloud adds the salary of the first tuple to the first values of the attributes A_x and A_y with o_1. Then, the cloud multiplies the summation outputs by o_1.

For the second tuple, the cloud performs the same operations, as did on the first tuple; however, the string-matching resultant o_2 becomes 0, which results in the second values of the attributes A_x and A_y to be 0. The cloud performs the same operations on the remaining tuples. Finally, the cloud sends the summation of o_i (*i.e.*, 2), the sum of the salaries of qualified tuples (*i.e.*, 6000), and the sum of outputs of the functions f_1 and f_2 (*i.e.*, 6004), to the user. Note that for this query, $Isum_{f1f2} - 2 \times Isum_q = Isum_\ell$, *i.e.*, $6004 - 2 \times 2 = 6000$.

Correctness. Now, we show why does the sum query result verification method work. We assume that the occurrences of the qualified tuples against a query predicates can be verified using the method given in §IV-A. Consider two cases: (*i*) all the clouds discard an entire identical tuple for processing, or

[10] If the user is interested, the user can also verify this result using the method given in §IV-A.

Dept	Salary	o values	A_x and f_1	A_y and f_2
Testing	1000	1	1(**200**+1000+1)=1201	1(−**1200**+1000+1)= −199
Security	100000	0	0(**1000**+100000+0)=0	0(−**101000**+100000+0)=0
Testing	5000	1	1(−**5900**+5000+1)= −899	1(**900**+5000+1)=5901
Design	2000	0	0(**2000**+2000+0)=0	0(−**4000**+2000+1)=0
Design	1500	0	0(**500**+1500+0)=0	0(−**2000**+1500+0)=0
Design	2000	0	0(−**2100**+2000+0)=0	0(**100**+2000+0)=0
		2	$\sum f_1$ =302	$\sum f_2$ =5702

Figure 4: An execution of the sum query verification.

(*ii*) all the clouds correctly process the query predicate, but they discard the i^{th} values of the attributes A_ℓ, A_x, and A_y.

The first case is easy to deal with, since the count query verification will inform the user that an identical tuple is discarded by the cloud for any processing. In the second case, the user finds $Isum_{f1f2} - 2 \times Isum_q \neq Isum_\ell$. The reason is as follows: An adversary cannot provide a wrong value of $Isum_q$, since it is detected by count query verification. In order to hold the equation $Isum_{f1f2} - 2 \times Isum_q = Isum_\ell$, the adversary needs to generate shares such that $Isum_{f1f2} - Isum_\ell = 2 \times Isum_q$, and an adversary cannot generate any share, since the adversary cannot collude the majority of the clouds.

VI. Maximum

This section provides methods for finding the maximum value and retrieving the corresponding tuples for the two types of queries, where the first type of query (`QMax1`) does not have any query condition, while another (`QMax2`) is a conditional query, as follows:

```
QMax1. select *, MAX(salary) from Employee
QMax2. select *, MAX(salary) from Employee
         where E.Dept= 'Testing'
```
[11]

Note that the strongly secure string-matching secret-sharing algorithms (as explained in §II) cannot find the maximum value, as these algorithms provide only equality checking mechanisms, not comparing mechanisms to compare between values. For answering maximum queries, we provide two methods: The first method, called `SDBMax` is applicable for the case when only a single DB owner outsources the database. It will be clear soon that `SDBMax` takes only one communication round when answering an unconditional query (like `QMax1`) and at most two communication rounds for answering a conditional query (like `QMax2`). The second method, called `MDBMax` is applicable to the scenario when multiple DB owners outsource their data to the clouds.

`SDBMax`. In this section, we assume that A_c be an attribute of the relation $S(R^1)$ on which the user wishes to execute a maximum query. Our main idea is based on a combination of order-preserving secret-sharing (OP-SS) [12], [17], [24] and SSS [6], [21], [24], [25] techniques. Specifically, for answering maximum queries, `SDBMax` uses the two relations $S(R^1)$ and $S(R^2)$, which are secured using secret-shared and OP-SS, respectively, as explained in §III-A. In particular, according to our data model (§III-A), the attribute A_c will exist in the relations $S(R^1)_i$ and $S(R^2)_i$ at the cloud i. The strategy is to jointly execute a query on the relations $S(R^1)_i$ and $S(R^2)_i$ and obliviously retrieve the entire tuple from $S(R^1)_i$. In this paper, due to space restrictions, we develop `SDBMax` for the case when only a single tuple has the maximum value; for example, in `Employee` relation (see Figure 1), the maximum salary over all employees is unique.

A. Unconditional Maximum Query

Recall that by observing the shares of the attribute A_c of the relation $S(R^1)$, the cloud cannot find the maximum value of the attribute A_c. However, the cloud can find the maximum value of the attribute A_c using the relation $S(R^2)$, which is secret-shared using OP-SS. Thus, to retrieve a tuple having the maximum value in the attribute A_c of the relation $S(R^1)_i$, the i^{th} cloud executes the following steps:

1) *On the relation* $S(R^2)_i$. Since the secret-shared values of the attribute A_c of the relation $S(R^2)_i$ are comparable, the cloud i finds a tuple $\langle S(t_k), S(value)\rangle_i$ having the maximum value in the attribute A_c, where $S(t_k)_i$ is the k^{th} secret-shared tuple id (in the attribute `SSTID`) and $S(value)_i$ is the secret-shared value of the A_c attribute in the k^{th} tuple.
2) *On the relation* $S(R^1)_i$. Now, the cloud i performs the join of the tuple $\langle S(t_k), S(value)\rangle_i$ with all the tuples of the relation $S(R^1)_i$ by comparing the tuple ids (`TID` attribute's values) of the relation $S(R^1)_i$ with $S(t_k)_i$, as follows:

$$\textstyle\sum_{k=1}^{k=n} A_p[S(a_k)]_i \times (\mathtt{TID}[S(a_k)]_i \otimes S(t_k)_i)$$

Where p ($1 \leq p \leq m$) is the number of attributes in the relation R and `TID` is the tuple id attribute of $S(R^1)_i$. To say, the cloud i compares the tuple id $\langle S(t_k)\rangle_i$ with each k^{th} value of the attribute `TID` of $S(R^1)_i$ and multiplies the resultant by the first m attribute values of the tuple k. Finally, the cloud i adds all the values of each m attribute.

Correctness. Firstly, we know that the cloud i can find the tuple having the maximum value in the attribute A_c of the relation $S(R^2)_i$. Afterward, the comparison of the tuple id $S(t_k)_i$ with all the values of the `TID` attribute of the relation $S(R^1)_i$ results in $n-1$ zeros (when the tuple ids do not match) and only one (when the tuple ids match) of secret-share form. Further, the multiplication of the resultant (0 or 1 of secret-share form) by the entire tuple will leave only one tuple in the relation $S(R^1)_i$, which satisfies the query.

Information leakage discussion. The adversary will know only the order of the values, due to OP-SS implemented on the relation $S(R^2)$. However, revealing only the order is not threatening, since the adversary may know the domain of the values, for example, the domain of age or salary.

Recall that, as mentioned in §III-A, the relations $S(R^1)$ and $S(R^2)$ share attributes: `TID`/`SSTID` and A_c (the attribute on which a comparison operation will be carried). However, by just observing these two relations, the adversary cannot know any relationship between them, as well as, which tuple

[11]Note that we showed only a single dimensional condition in `QMax2` query. Our proposed algorithms (without any modification) can find maximum/minimum while satisfying conjunctive and disjunctive conditions.

of the relation $S(R^1)$ has the maximum value in the attribute A_c, due to different representations of common TID/SSTID and A_c values between the relations. Furthermore, after the above-mentioned maximum query (QMax1) execution, the adversary cannot learn which tuple of the relation $S(R^1)$ has the maximum value in the attribute A_c, due to executing an identical operation on each tuple of $S(R^1)$ when joining with a single tuple of $S(R^2)$.

Complexities. For answering unconditional maximum queries (QMax1) when only one tuple has the maximum value in the attribute A_c, SDBMax requires only one communication round between the user and each cloud. The cloud has to scan the relation $S(R^1)$ only once due to the join operation. Further, since the maximum value is looked up using the cleartext tuple ids (CTID attribute), an index structure is used over the CTID attribute to avoid a linear scan.

B. Conditional Maximum Query

The maximum value of the attribute A_c may be different from the A_c's maximum value of the tuple satisfying the where clause of a query. For example, in Employee relation, the maximum salary of the testing department is 2000, while the maximum salary of the employees is 100000. Thus, the method given for answering unconditional maximum queries is not applicable here. In the following, we provide a method to answer maximum queries that have conditional predicates (like QMax2), and that uses *two* communication rounds between the user and the clouds, as follows:

Round 1. The user obliviously knows the tuple ids of the relation $S(R^1)$ satisfying the where clause of the query (the method for obliviously finding the tuple ids is given below).

Round 2. The user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the clouds. Each cloud i finds the maximum value of the attribute A_c in the requested tuple ids by looking into the attribute CTID of the relation $S(R^2)_i$ and results in a tuple, say $\langle S(t_k), S(value)\rangle_i$, where $S(t_k)_i$ shows the secret-shared tuple id (from SSTID attribute) and $S(value)_i$ shows the secret-shared maximum value. Now, the cloud i performs a join operation between all the tuples of $S(R^1)_i$ and $\langle S(t_k), S(value)\rangle_i$, as performed when answering unconditional maximum (QMax1) queries; see §VI-A. This operation results in a tuple that satisfies the conditional maximum query.

Note. The difference between the methods for answering unconditional and conditional maximum queries is that first we need to know the desired tuple ids of the $S(R^1)$ relation satisfying the where clause of a query in the case of conditional maximum queries.

Correctness. The correctness of the above method can be argued in a similar manner as the method for answering unconditional maximum queries.

Information leakage discussion. In round 1, due to obliviously retrieving the tuple ids of $S(R^1)$, the adversary cannot know which tuples satisfy the query predicate. In round 2, the user sends only the desired tuple ids in cleartext to fasten the lookup of OP-SS maximum salary. Note that by sending cleartext tuple ids, the adversary learns the number of tuples satisfy the query predicate;[12] however, the adversary cannot learn which tuples of the relation $S(R^1)$ have those tuple ids. Due to OP-SS, the adversary also knows only the order of values of the attribute A_c in the requested tuple ids. Note that the joining the tuple of $S(R^2)$, which has the maximum value in the attribute A_c, with all tuples of $S(R^1)$ will not reveal which tuple satisfies the query predicate as well as have the maximum value in A_c.

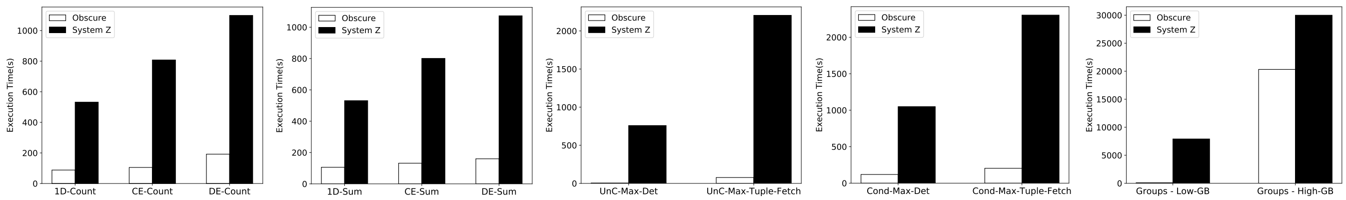
Complexities. For answering conditional maximum queries (QMax2) when only one tuple has the maximum value in the attribute A_c, SDBMax requires two communication rounds between the user and each cloud. The cloud has to scan the relation $S(R^1)$ twice, due to first finding tuples satisfying the query predicate and, then, join operation.

Aside: Hiding frequency-analysis in round 2 used for conditional maximum queries. In the above-mentioned round 2, the user reveals the number of tuples satisfying a query predicate, and now, we provide a method to hide frequency-count information, as follows:

User-side. The user interpolates the received tuple ids (after round 1) and sends the desired tuple ids with some fake tuple ids, which do not satisfy the query predicate in the round 1, in cleartext to the clouds. Let $x = r + f$ be the tuple ids that are transmitted to the clouds, where r and f be the real and fake tuple ids, respectively. Note that the maximum value of the attribute A_c over x tuples may be more than the maximum value over r tuples. Hence, the user does the following computation to appropriately send the tuple ids: The user arranges the x tuple ids in a $\sqrt{x} \times \sqrt{x}$ matrix, where all r real tuple ids appear before f fake tuple ids. Then, the user creates $\sqrt{x}$ groups of tuples ids, say $g_1, g_2, \ldots, g_{\sqrt{x}}$, where all tuples ids in an i^{th} row of the matrix become a part of the group g_i. Note that in this case only one of the groups, say g_{mix}, may contain both the real and fake tuple ids. Now, the user asks the cloud to find the maximum value of the attribute A_c in each group except for the group g_{mix} and to fetch all $\sqrt{x}$ tuples of the group g_{mix}.

Cloud-side. For each group, g_j, except the group g_{mix}, each cloud i finds the maximum value of the attribute A_c by looking into the attribute CTID of the relation $S(R^2)_i$ and results in a tuple, say $\langle S(t_k), S(value)\rangle_i$. Further, the cloud i fetches all $\sqrt{x}$ tuples of the group g_{mix}. Then, the cloud i performs a join operation (based on the attribute TID and SSTID, as performed in the second step for answering unconditional maximum queries; see §VI-A) between all the tuples of $S(R^1)_i$ and $2\sqrt{x}-1$ tuples obtained from the relation $S(R^2)$, and returns $2\sqrt{x}-1$ tuples to the user. The user finds the maximum value over the r real tuples. Note that $2\sqrt{x}-1$ tuples must satisfy a conditional maximum query; however,

[12] The adversary may already know the classification of tuples based on some criteria, due to her background knowledge. For example, the number of employees working in a department or the number of employees of certain names/age. Hence, revealing the number of tuples satisfying a query does not matter a lot; however, revealing that which tuples satisfy a query may jeopardize the data security/privacy.



(a) Count query. (b) Sum query. (c) Maximum query 1. (d) Maximum query 2. (e) Group-by query.

Figure 5: Exp 1. Effectiveness of OBSCURE.

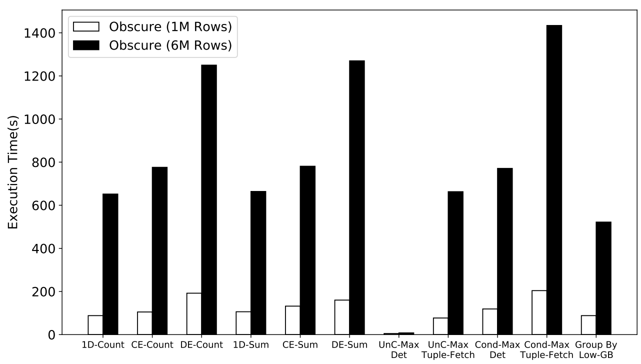
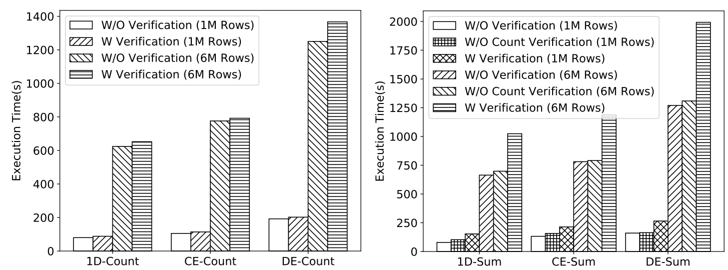


Figure 6: Exp 2. Scalability.



(a) Exp 3: Count query. (b) Exp 3: Sum query.

Figure 7: Exp 3. Result verification.

due to space restrictions, we do not prove this claim here.

Note that this method, on one hand, hides the frequency-count; on the other hand, it requires the clouds and the user process more tuples than the method that reveals the frequency-count.

Obliviously finding the tuple ids. For finding the tuple ids, each cloud k executes the following operation: $\texttt{Index}[i]_k \times (A_p[i]_k \otimes S(v)_k)$, *i.e.*, the cloud executes string-matching operations on each value of the desired attribute, say A_p, of the relation $S(R^1)$ and checks the occurrence of the query predicate v. Then, the cloud k multiplies the i^{th} resultant of the string-matching operation by the i^{th} value of `Index` attribute of the relation $S(R^1)$. Finally, the cloud sends all the n values of the attribute `Index` to the user, where n is the number of tuples in the relation. The user interpolates the received values and knows the desired tuple ids. In Appendix B, we provide two methods for knowing the tuple ids.

VII. EXPERIMENTS

This section evaluates the above-mentioned algorithms against an industrial MPC-based system Z. Note that we refer to as system Z to hide its identity. We used LineItem table of TPC-H benchmark to generate the dataset of 1M and 6M rows that were inserted into MySQL database system. We used in-home machines that were running virtual machines (VMs), which we treated as clouds. We executed the following experiments:

Exp 1: Effectiveness of OBSCURE. To explore the effectiveness of OBSCURE, we tested OBSCURE and the system Z for count, sum, unconditional and conditional maximum, and group-by queries.

Count and sum queries. Figure 5 shows the time taken by OBSCURE and the system Z. Figures 5a and 5b show the time taken by one-dimensional (1D), conjunctive-equality (CE), and disjunctive-equality (DE) count and sum queries, respectively, when using OBSCURE and the system Z. Note that OBSCURE is at least 20% faster than the system Z, because the system Z requires inter-cloud communication to evaluate the function for each tuple.

Maximum queries. For the case of unique maximum value and unconditional maximum queries (`QMax1`, see §VI), Figure 5c shows that determining the maximum value (UnC-Max-Det) is very efficient due to OP-SS, as compared to the system Z. In addition, Figure 5c shows that fetching the tuple having the maximum value (UnC-Max-Tuple-Fetch) is also more efficient when using OBSCURE as compared to the system Z. Figure 5d shows the time taken by a conditional maximum query (`QMax2`, see §VI) for finding the maximum value (Cond-Max-Det) and fetching the maximum tuple (Cond-Max-Tuple-Fetch). Note that in both the cases, we save significant time due to OP-SS, while also preventing background-knowledge-based attacks on OP-SS.

Group-by queries. Though we have not formally defined a group-by query. The reason to include the group-by query in the experiment is that, in short, a group-by query works in a similar manner like a one-dimensional sum query. Figure 5e shows the time taken by a group-by query when the number of groups was 7 (Low-GB, due to Linenumber attribute that has seven values) and the number of groups was 10K (High-GB, due to Suppkey that has 10K values). In both the group-by queries, OBSCURE performed better than the system Z.

Exp 2. Scalability of OBSCURE. In order to validate the behavior of OBSCURE on a large-sized data, we executed all the operations on the LineItem table having 6M rows; see Figure 6. Note that previous work [13] on secret-sharing-based aggregation queries used only 1000 rows. We do not compare OBSCURE against the system Z, because inserting 6M rows in the system Z is a cumbersome task, which will be clear in Exp 4. It is clear from Figure 6 that the time taken by OBSCURE increases linearly as the dataset size increases, since OBSCURE scans each tuple of the dataset.

Exp 3. Overheads of result verification. From this experiment, it is clear that our non-mandatory result verification steps do not incur a significant cost at the cloud in the case of count query (see Figure 7a). However, in case of a sum query, the cost increases, due to verifying the count results and the sum query. However, if one drops count query result verification, the cost decreases significantly (see Figure 7b).

Exp 4. Data generation. Recall that for this paper, we used a unary representation of the data (§II), which can also be compacted using binary representation. In our last experiment (see Table II), we shows the time and size to generate LineItem table of size 1M rows in OBSCURE and in the system Z. Note that due to unary representation, the size of the data increases significantly as compared to the system Z; however, having such a huge dataset, the data generation (and query processing) time of OBSCURE is significantly less than the system Z.

Systems	Time	Size (in GB)
OBSCURE	$\approx$ 1.3 hours	$\lvert S(R^1)\rvert = 2.60$, $\lvert S(R^2)\rvert = 0.669$
System Z	$\approx$ 9 hours	1.01

Table II: Exp. 4. Data generation.

VIII. Other Operations for Maximum Queries

A. Verification of Maximum Query

Now, we develop a method to verify the tuple having maximum value in an attribute A_c is correct. Note that verifying only the maximum value is trivial. The reason is as follows: when the cloud joins the single tuple of the relation $S(R^2)$ with all the tuples of the relation $S(R^1)$, the cloud is not aware of x, and the joining tuple also has the same value in a different SS format. Sending the entire tuple and the tuple from $S(R^1)$ will make sure that the maximum value is identical. The cloud can never change both the values, unless the adversary colludes the majority of the clouds.

Verification of retrieved tuple. Thus, we provide a method to verify that the returned tuple from the cloud is correct. Note that this method is an extension of the sum verification method. The cloud computes two functions, f_1 and f_2, to verify both the conditions of sum-query verification in an oblivious manner, as follows:

$$op_1 = f_1(x) = \sum_{i=1}^{i=n} o_i(x_i + s_i)$$

$$op_2 = f_1(x) = \sum_{i=1}^{i=n} o_i(y_i + s_i)$$

i.e., the cloud performs the functions f_1 and f_2 on some n values, (it will be clear soon what are these n values). In the above equations, o_i is the output of the string-matching operation carried on the i^{th} value of the attribute A_q, and a_i be the i^{th} $(1 \leq i \leq n)$ value of the attribute A_ℓ. The cloud sends the difference of the outputs of the function f_1, denoted by op_1, and the outputs of f_2, denoted by op_2, to the user. Particularly, the verification method for a sum query works as follows:

The DB owner. Analogous with the count and sum verification methods, if the DB owner wants to provide verification for tuple retrieval queries, new attributes, say A_x and A_y, should be added along with one value to each of the attribute values of a tuple. Here, first, we describe what is the new value added to each attribute value, and then, describe what are the values of the attributes A_x and A_y.

Let A_1 be an attribute having only numbers. For the attribute A_1, the newly added i^{th} value in cleartext is same as the existing i^{th} value in the attribute A_1. Let A_2 be an attribute having English alphabets, say the attribute `Name` in Employee relation in Figure 1. The new value is the sum of the positions of the value; for example, the first value in the attribute `Name` is `John`, the DB owner adds 47 (10+15+8+14). When creating shares of the two values at the i^{th} position of the attribute A_1 or A_2, the first value's shares are created using the mechanism that support string-matching at the cloud without involving the user, as mentioned in §II, and the second value's shares are created using SSS.

The i^{th} values of the attributes A_x and A_y are any two arbitrary random numbers whose difference equals to $-a_i$, where a_i is the i^{th} value obtained after summing all the newly added values to each attribute. The values of the attributes A_x and A_y are also secret-shared using SSS. For example, in Figure 8, numbers show newly added values to attributes, sum of these values, and random numbers (in bold-face) of the attributes A_x and A_y in cleartext.

Clouds. Each cloud k executes the private string-matching operation on the i^{th} $(1 \leq i \leq n)$ value of the attribute `TID` against a requested tuple id. The string-matching resultant is multiplied to each of the m values of the i^{th} tuple. Note that this steps results in only one tuple left after performing string-matching operation on each tuple. Then, for the i^{th} tuple, the cloud k adds all the m values that are appended for verification purpose. The sum of the m values is denoted by s_i. Afterward, the cloud k executes the functions f_1 and f_2 on each value x_i and y_i of the attributes A_x and A_y, by following the above-mentioned equations. Finally, the cloud k sends the following three things to the user: (*i*) the sum of the each attribute value, say $\langle sum_{Ai}\rangle_k$, where $1 \leq i \leq m$; (*ii*) the difference of outputs of the functions f_1 and f_2, say $\langle \mathit{diff}_{f1f2}\rangle_k$.

User-side. The user interpolates the received two values from each cloud, which results in $Isum_{Ai}$ and $I\mathit{diff}_{f1f2}$. The user creates values for all such attributes that have English alphabet, like the DB owner does, and compares against $Isum_{f1f2}$, and if it finds equal, the cloud has correctly send the tuple.

Example. Figure 8 shows the newly added values to the each attribute (denoted with a $'$ symbol to distinguish them from the original attribute name) of the relation Employee and the values of the attribute A_x and A_y. We show the

EmpID′	Name′	Salary′	Dept′	TID	o	A_x	A_y
106	47	1000	80	3	1	1(**500**+1233)=1733	1(**-733**+1233)=500
106	47	100000	120	2	0	0(**400**+100273)=0	0(**-99873**+100273)=0
107	19	5000	80	5	0	0(**200**+5211)=0	0(**-5011**+5211)=0
108	32	2000	51	4	0	0(**600**+2195)=0	0(**-1595**+2195)=0
109	30	1500	51	1	0	0(**300**+1690)=0	0(**-1390**+1690)=0
110	38	2000	51	6	0	0(**100**+2199)=0	0(**-2099**+2199)=0
						$op_1 = 1733$	$op_2 = 500$

Figure 8: An execution of the tuple retrieval verification.

verification process for the first tuple id. Note that the values and computation is shown in the cleartext; however, at the cloud, the values are of secret-share form and the computation will be carried on shares.

B. Multiple Occurrences of the Maximum Value

In practical applications, more than one tuple may have the maximum value in an attribute, *e.g.*, two employees (E103 and E015) earn the maximum salary in design department; see Figure 1. However, the above-mentioned methods (for QMax1 or QMax2) cannot fetch all those tuples from the relation $S(R^1)$ in one round. The reason is that since the cloud i uses OP-SS values of the attribute A_c of the relation $S(R^2)_i$ for finding the maximum value, where more than one occurrences of a value have different representations, the cloud i cannot find all the tuples of $S(R^2)$ having the identical maximum value, by looking OP-SS values.

In this subsection, we, thus, provide a simple two-communication-round method for solving unconditional maximum queries. This method can be easily extended to conditional maximum queries.

Data outsourcing. The DB owner outsources the relation $S(R^1)$ as mentioned in §III-A. However, the DB owner outsources the relation $S(R^2)$ with four columns: CTID, SSTID, OP-SS-A_c, and SS-A_c. The first three columns are created in the same way as mentioned in §III-A. The i^{th} value of SS-A_c attribute has the same value as the i^{th} value of OP-SS-A_c attribute. However, this value is secret-shared using the unary representation, as the column A_c of the relation $S(R^1)$ has. However, the DB owner uses different polynomial over the i^{th} value of the attribute A_c of $S(R^1)$ and the attribute SS-A_c of $S(R^2)$, so that the adversary cannot related two relations.

Query execution. The method uses two communication rounds as follows:

Round 1. In round 1, the cloud i finds a tuple $\langle S(t_k), S(value_1), S(value_2)\rangle_i$ having the maximum value (denoted by $\langle S(value)_1\rangle_i$) in the attribute A_c, where $S(t_k)_i$ is the k^{th} secret-shared tuple id (in the attribute SSTID) and $\langle S(value)_2\rangle_i$ is the secret-shared value of the SS-A_c attribute in the k^{th} tuple. Afterward, the cloud i performs the following:

$$\texttt{Index}[k] \times (A_C[S(k)]_i \otimes S(value_2)_i), 1 \leq k \leq n$$

i.e., the cloud compares $\langle S(value_2)\rangle_i$ with each k^{th} value of the attribute A_c of the relation $S(R^1)$ and multiplies the resultant by the k^{th} index values. The clouds i provides a list of n numbers to the user.

Round 2. After interpolating n numbers, the user gets a list of n numbers having 0 and Index values, where the maximum value of the attribute A_c exists. Then, the user fetches all the tuples having the maximum values based on the received Index value. In particular, the user creates new secret-shares of the matching indexes in a way that the cloud can perform searching operation on TID attribute. The cloud executes the following computation to retrieve all the tuples, say $\mathcal{T}$, having the maximum value in the attribute A_c:

$$\textstyle\sum_{k=1}^{k=n} A_p[S(a_k)]_i \times (\texttt{TID}[S(a_k)]_i \otimes S(t_j)_i)$$

Where $1 \leq p \leq m$, $1 \leq j \leq \mathcal{T}$ and $1 \leq k \leq n$, i.e., the cloud i compares each received tuple id $\mathcal{T}$ with each tuple id of the relation $S(R^1)_i$ and multiplies the resultant to the first m attributes of the relation $S(R^1)_i$. Finally, the cloud i adds all the attribute values for each tuple id $\mathcal{T}$.

Complexities. As mentioned, fetching all tuples having the maximum value in the attribute A_c requires two communication rounds when answering an unconditional query. Further, each cloud scans the entire relation $S(R^1)$ twice. However, finding the maximum number over the attribute OP-SS-A_c can be done using an index.

Information leakage discussion. The adversary learns the order of the values. The adversary will not learn which tuple has the maximum value in the attribute A_c. But, the adversary may learn how many tuples have the maximum value. This can be preventing by asking queries for fake tuples.

Aside. If we want to increase one communication round, then there is no need to outsource the relation $S(R^2)$ as suggested above. Instead of that the cloud provide a tuple having the maximum value in the attribute A_c, and then, the user find occurrences of the maximum value in one additional round.

Note. Answering conditional maximum query. We are not providing details for fetching all the tuples having the maximum in the case of conditional maximum queries. For answering a conditional maximum query, the user include the above-mentioned two steps to the method given in §VI-B. Thus, fetching all tuples having the maximum value in the attribute A_c requires three communication rounds, and each cloud scans the entire relation $S(R^1)$ three times. In particular, in the first round, the cloud i provides Index values to the user. In the second round, the cloud i finds the tuple having the maximum value in the attribute A_c from the requested tuple ids, implements the above-mentioned method given in round 1, and provides a list of n numbers. In the last round, the user fetches all the desired tuples.

C. Finding Maximum over Datasets Outsourced by Multiple DB Owners

In this section, we explain a method, named `MDBMax` for the case when multiple DB owner outsource their data to clouds, e.g., smart meters. Note that for the case of multiple DB owners, `SDBMax` method cannot work, as different DB owners do not share any information for creating OP-SS. We describe `MDBMax` for a list, say A_c, having n numbers outsourced by k DB owner/devices, where $k \leq n$.

Data outsourcing. Consider that an i^{th} DB owner wishes to outsource a number, say v. The i^{th} DB owner creates shares of v using a secret-sharing mechanism (either HSS or AA) that allows string-matching operations at the cloud and sends to the c non-communicating clouds, as described in §III-A. However, note that, here, we do not outsource numbers using the unary representation, which was used for other queries in previous sections. In this case, the DB owner first creates a binary representation of the number and then creates the shares. Binary representation allows us to execute 2's complement-based signbit computation, as follows:

Query execution. `MDBMax` uses 2's complement-based signbit computation for each pair of shares at a cloud. The cloud j considers an i^{th} ($1 \leq i \leq n$) share as the maximum value and compares the i^{th} share against the remaining $n-1$ shares.

Thus, for each number at the i^{th} position, say V_i, the cloud j computes the signbit with all the other numbers using 2's complement-based subtraction, *i.e.*, $signbit(V_i - V_x)$, $x \neq i$, and $1 \leq x, i \leq n$. Recall that the signbit results in 1 of secret-share form, if $V_i < V_x$; otherwise, 0. Then, the cloud j adds all $n-1$ signbit values computed for the i^{th} share of the list A_c. Therefore, after comparing each pair of inputs and adding corresponding $n-1$ signbit values, the cloud j has a vector, say vec, of n shares. The user asks the count query (§IV) to find the occurrences of 0 in vec (it will be clear soon why the user is asking for counting 0) and the sum of the values of A_c for which the count query resulted in 1 of secret-shared form.

Example. The following table shows how does the cloud find the maximum value without using OP-SS. Note that for the purpose of explanation, we use cleartext values and computations; however, the cloud will perform all operations over secret-shared numbers. The list A_c contains five numbers: 10, 20, 90, 50, and 90. Note that the sum of signbit for the maximum value is 0. The cloud executes the count query for the value of 0, multiplies the i^{th} resultant to the i^{th} value of A_c, and sends the sum of the count query results and the sum of values of A_c after multiplication. The user receives 2 and 180 as the output of the count and sum queries, respectively, and so that the user knows the maximum value is 90.

A_c	Signbits					Sum of signbits	String-matching result	Maximum value
	10	20	90	50	90			
10	0	1	1	1	1	4	0	0
20	0	0	1	1	1	3	0	0
90	0	0	0	0	0	0	1	90
50	0	0	1	0	1	2	0	0
90	0	0	0	0	0	0	1	90
Answers to the count and sum queries							2	180

Complexities. `MDBMax` requires n^2 comparisons and $2n+1$ scan rounds of the list A_c, where the first n rounds are used in comparing each pair of numbers, other n rounds are used for adding $n-1$ signbits for each number, and one additional round for executing count and sum queries.

Minimum queries over numbers outsourced by multiple DB owners. Here, we also compare each pair of numbers. However, for each number at the i^{th} position, say V_i, we compute the signbit with all the other numbers using 2's complement-based subtraction, as follows: $signbit(V_x - V_i)$, $x \neq i$, and $1 \leq x, i \leq n$. As a result, after adding $n-1$ signbits for each number, the minimum values has 0, and the user asks for the count query for 0 and the sum of the values of A_c for which the count query resulted in 1 of secret-shared form.

IX. Conclusion

We proposed information-theoretic secure and communication efficient aggregation queries (count, sum, and maximum having single dimensional, conjunctive, or disjunctive query predicates) on a secret-shared dataset outsourced by either a single DB or multiple DB owners. We considered malicious adversarial cloud when all (or possibly the majority of) the clouds deviate from the algorithm in an identical manner, due to software/hardware bugs. Thus, we proposed efficient result verification algorithms. Further, we evaluated our algorithms against an industrial MPC-based secure database and found that our algorithms work significantly better than the MPC-based secure database.

References

[1] H. Hacigümüs *et al.*, "Executing SQL over encrypted data in the database-service-provider model," in *SIGMOD*, pp. 216–227, 2002.
[2] S. Goldwasser *et al.*, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984.
[3] D. X. Song *et al.*, "Practical techniques for searches on encrypted data," in *IEEE SP*, pp. 44–55, 2000.
[4] C. Gentry, *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
[5] R. Agrawal *et al.*, "Order-preserving encryption for numeric data," in *SIGMOD*, pp. 563–574, 2004.
[6] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[7] A. Beimel, "Secret-sharing schemes: A survey," in *IWCC*, pp. 11–46, 2011.
[8] https://shattered.io/.
[9] D. W. Archer *et al.*, "From keys to databases - real-world applications of secure multi-party computation," *IACR Cryptology ePrint*, 2018.
[10] Stealth Pulsar, available at:http://www.stealthsoftwareinc.com/.
[11] D. Bogdanov *et al.*, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, vol. 5283, pp. 192–206, 2008.
[12] F. Emekçi *et al.*, "Privacy preserving query processing using third parties," in *ICDE*, p. 27, 2006.
[13] F. Emekçi *et al.*, "Dividing secrets to secure data outsourcing," *Inf. Sci.*, vol. 263, pp. 198–210, 2014.
[14] S. Dolev *et al.*, "Privacy-preserving secret shared computations using mapreduce," *CoRR*, vol. abs/1801.10323, 2018.
[15] W. Jiang *et al.*, "Transforming semi-honest protocols to ensure accountability," *Data Knowl. Eng.*, vol. 65, no. 1, pp. 57–74, 2008.
[16] B. Thompson *et al.*, "Privacy-preserving computation and verification of aggregate queries on outsourced databases," in *PETS*, pp. 185–201, 2009.
[17] M. A. Hadavi *et al.*, "AS5: A secure searchable secret sharing scheme for privacy preserving database outsourcing," in *DPM*, pp. 201–216, 2012.
[18] H. Corrigan-Gibbs *et al.*, "Prio: Private, robust, and scalable computation of aggregate statistics," in *NSDI*, pp. 259–282, 2017.
[19] M. Burkhart *et al.*, "SEPIA: privacy-preserving aggregation of multi-domain network events and statistics," in *USENIX*, pp. 223–240, 2010.
[20] I. Damgård *et al.*, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *TCC*, pp. 285–304, 2006.
[21] S. Dolev *et al.*, "Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract," in *SCC*, pp. 21–29, 2015.

[22] H. Avni *et al.*, "SSSDB: database with private information search," in *ALGOCLOUD*, pp. 49–61, 2015.
[23] R. M. Corless and N. Fillion, "A graduate introduction to numerical methods," *AMC*, vol. 10, p. 12, 2013.
[24] E. Boyle *et al.*, "Homomorphic secret sharing: Optimizations and applications," in *CCS*, pp. 2105–2122, 2017.
[25] E. Boyle *et al.*, "Foundations of homomorphic secret sharing," in *ITCS*, pp. 21:1–21:21, 2018.
[26] S. Dolev *et al.*, "Secret shared random access machine," in *ALGOCLOUD*, vol. 9511, pp. 19–34.
[27] R. Canetti *et al.*, "Adaptively secure multi-party computation," in *STOC*, pp. 639–648, 1996.
[28] C. Wang *et al.*, "Secure ranked keyword search over encrypted cloud data," in *ICDCS*, pp. 253–262, 2010.
[29] S. Yu *et al.*, "Attribute based data sharing with attribute revocation," in *ASIACCS*, pp. 261–270, 2010.
[30] M. Castro *et al.*, "Practical byzantine fault tolerance," in *OSDI*, 1999.
[31] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.
[32] M. J. Freedman *et al.*, "Keyword search and oblivious pseudorandom functions," in *TCC*, pp. 303–324, 2005.
[33] C. Chu *et al.*, "Efficient *k*-out-of-*n* oblivious transfer schemes with adaptive and non-adaptive queries," in *PKC*, pp. 172–183, 2005.

Appendix

Appendix A
Minimum, Top-k, and Reverse Top-k

In this section, we focus on the minimum and top-k/reverse-top-k finding algorithms on an attribute, say A_c. Further, we assume that any value in the attribute A_c appears only once.

Minimum. Consider the following two queries `QMin1` (unconditional minimum) and `QMin2` (conditional minimum).

```
QMax1. select *, MIN(salary) from Employee
QMax2. select *, MIN(salary) from Employee
         where E.Dept= 'Testing'
```

Here, in short, we explain how to execute these queries on the relations $S(R^1)$ and $S(R^2)$, since these queries are similar to maximum queries §VI. To execute an unconditional minimum query, the user follows the same strategy for solving `QMax1` (§VI-A); however, the user asks for the minimum value from the relation $S(R^1)$. First, each cloud i finds a tuple, say $\langle S(t_k), S(value)\rangle_i$, where $S(t_k)_i$ is the k^{th} secret-shared tuple id (in the attribute `SSTID`) and $S(value)_i$ is the secret-shared minimum value of the A_c attribute in the k^{th} tuple. Finally, the cloud i compares the tuple id $\langle S(t_k)\rangle_i$ with each k^{th} value of the attribute `TID` of $S(R^1)_i$ and multiplies the resultant by the first m attribute values of the tuple k. Finally, the cloud i adds all the values of each m attribute.

To execute a conditional minimum query, the user operates in two rounds, like a conditional maximum query; see §VI-B. In the round 1, the user obliviously knows the tuple ids of the relation $S(R^1)$ satisfying query predicate. In round 2, the user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the clouds. Each cloud i finds the minimum value of the attribute A_c in the requested tuple ids by looking into the attribute `CTID` of the relation $S(R^2)_i$ and results in a tuple, say $\langle S(t_k), S(value)\rangle_i$, where $S(t_k)_i$ shows the secret-shared tuple id (from `SSTID` attribute) and $S(value)_i$ shows the secret-shared minimum value. Finally, the cloud i performs a join operation between all the tuples of $S(R^1)_i$ and $\langle S(t_k), S(value)\rangle_i$, as performed when answering unconditional maximum (`QMax1`) queries; see §VI-A.

Correctness and information leakage. The correctness arguments and information leakage of a minimum query is similar to maximum queries.

Top-k. We again consider unconditional and conditional queries in the case of a top-k query. In both the cases, the user follows a similar approach, like maximum queries; see §VI; however, the user asks for top-k values instead of the maximum value.

Unconditional top-k query. To retrieve tuples having the top-k values in the attribute A_c of the relation $S(R^1)_i$, the i^{th} cloud executes the following steps:

1) *On the relation* $S(R^2)_i$. Since the secret-shared values of the attribute A_c of the relation $S(R^2)_i$ are comparable, the cloud i finds a set of k tuples, where k tuples have the top-k values in the attribute A_c. One of the k tuples is denoted by $\langle S(t_\ell), S(value)\rangle_i$, where $S(t_\ell)_i$ is the ℓ^{th} secret-shared tuple id (in the attribute `SSTID`) and $S(value)_i$ is the secret-shared value of the A_c attribute in the j^{th} tuple.
2) *On the relation* $S(R^1)_i$. Now, the cloud i performs the join of all the top-k tuples with all the tuples of the relation $S(R^1)_i$ by comparing the tuple ids (`TID` attribute's values) of the relation $S(R^1)_i$:

$$\sum_{j=1}^{j=n} A_p[S(a_j)]_i \times (\mathtt{TID}[S(a_j)]_i \otimes S(t_\ell)_i)$$

Where $1 \leq \ell \leq k$ and p ($1 \leq p \leq m$) is the number of attributes in the relation R and `TID` is the tuple id attribute of $S(R^1)_i$. To say, the cloud i compares each tuple id $\langle S(t_\ell)\rangle_i$ with each j^{th} value of the attribute `TID` of $S(R^1)_i$ and multiplies the resultant by the first m attribute values of the tuple j. Finally, the cloud i adds all the values of each m attribute.

Conditional top-k query. Answering conditional top-k queries require when all the values of the attribute A_c are unique requires two communication rounds between the user and the clouds, like a conditional maximum query, see §VI-B, as follows:
Round 1. The user obliviously knows the tuple ids of the relation $S(R^1)$ satisfying the query predicate.
Round 2. The user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the clouds. Each cloud i finds the top-k values of the attribute A_c in the requested tuple ids by looking into the attribute `CTID` of the relation $S(R^2)_i$ and results in a set of k tuples. Now, the cloud i performs a join operation between all the tuples of $S(R^1)_i$ and each of the k tuples of the relation $S(R^2)$, as performed above in answering an unconditional top-k query.
Note. A reverse-top-k query can also be executed in a same manner like top-k queries; however, the user asks for the minimum-k values.

Group-by. A group-by query finding the counting/sum of the values in any attribute can be executed like a conjunctive sum query; see §V. However, the user needs to know the number of the groups, and then, the user sends the names of the groups to the cloud on which the user wants to execute a group-by query. Note that it may reveal the number of groups in an attributes; and hence, the user may ask the query for some fake groups.

A group-by query finding the maximum/minimum can be also be done in a similar manner as executing conditional maximum queries; see §VI-B. However, here the user asks for the tuples ids for each group in round 1 of the query exsection, and then, asks to return a tuple having the maximum/minimum value in a specified attribute. Note that here the adversary again knows the number of groups and the number of tuples in each group. Hiding the number of tuples in a group can be done by following the method given in §VI-B, which returns $2\sqrt{x}-1$ tuples for a single group. Note that this method is only beneficial if the number of returned tuples for all the groups are significantly small than the number of tuples in the relation.

Appendix B
Methods for Finding Tuple Ids

A trivial solution for knowing the tuple ids satisfying a query predicate is given in §VI-B that transmits n numbers from each cloud to the user.

In the following method, we allow the adversary to know an upper bound on the number of tuples, say $\mathcal{T}$, satisfy the query predicate. We provide two methods: The first method executes $\mathcal{T}$ computations on each tuples and maintains $\mathcal{T}$ variables for each tuple. Thus, the cloud performs significant computations, when $\mathcal{T}$ is large. The second method is only applicable when less than $\sqrt{n}$ number of tuples satisfy a query. This methods maintains only three variable for each tuple.

The first method. The cloud creates $\mathcal{T}$ columns,[13] one for each tuple id that satisfies the query predicate, say v. Note that actually we do not need to create any column during implementation, we need to have $\mathcal{T}$ variables. For the purpose of explanation, we show $\mathcal{T}$ columns. Each column has allocated one of the values from 1 to $\mathcal{T}$ of secret-share form (provided by the user). After an oblivious computation over each tuple, if there are $\mathcal{T}$ occurrences of v, then each of the $\mathcal{T}$ columns will have one of the exact tuple id where v occurs. The cloud executes the following operation:

$$r \times o[1-(\mathit{signbit}(x-a)+\mathit{signbit}(a-x))]$$

Where r is the tuple id; $o = A_\ell[S(a_i)] \otimes S(v)$, $1 \leq i \leq n$, *i.e.*, the resultant output of matching the predicate v with each value of the attribute A_ℓ; $a = \sum_{i=1}^{i=n} A_\ell[S(a_i)] \otimes S(v)$, *i.e.*, the accumulated counting of the predicate v in the attribute A_ℓ; and x ($1 \leq x \leq \mathcal{T}$) is a value of the column, created for storing the tuple id.

Details. For r^{th} ($1 \leq r \leq n$) value of the attribute A_ℓ, the cloud executes counting operations for finding the occurrences of v in A_ℓ. The occurrences of v in the above-equation is denoted by a. For each resultant a, the cloud compares a against each of the $\mathcal{T}$ values using 2's complement method (as given in §II). The occurrence of v matches with only one of the $\mathcal{T}$ values, and thus, $\mathit{signbit}(x-a)+\mathit{signbit}(a-x)$ results in 0, *i.e.*, the difference of signbits of comparing two identical numbers is 0. For all the other subtraction, it will be either 1 or -1 of secret-share form. Note that for all the values of $\mathcal{T}$ that do not match with a, the above-mentioned equation will be 0 of secret-share form.

Since for the occurrence of v matching with one of the values of $\mathcal{T}$, $\mathit{signbit}(x-a)+\mathit{signbit}(a-x)$ results in 0, we subtract it from 1 to keep 1 on which we can multiply the tuple id r. Thus, if the tuple r has v in the attribute A_ℓ, the cloud keeps r to one of the $\mathcal{T}$ columns. It is important to note that if the r^{th} tuple has v in attribute A_ℓ and $(r+1)^{th}$ tuple do not have v in attribute A_ℓ, the value of accumulated count, a, will be same for the tuples r^{th} and $(r+1)^{th}$. Hence, the cloud may also keep the $(r+1)^{th}$ tuple id in the same column where it has kept r^{th} tuple id. In order to prevent this, we also multiply the result of the string-matching operation (denoted by o, see the above equation. Thus, the $(r+1)^{th}$ tuple id will not be stored. Finally, the cloud performs the addition operations on each $\mathcal{T}$ column and sends the final sum of each column to the user.

<u>*Example.*</u> Table III shows an implementation of tuple id finding method in cleartext to know the tuple ids that have `Dept = Testing`; see Figure 2a for Employee relation. Note that for each row, we perform string-matching operations whose results are stored in the variable o, and all the occurrences of the query predicate are stored in the variable a. The user asks the cloud to create two columns ($\mathcal{T}=2$) for keeping tuple ids.

For the first tuple, the string-matching operation results in $o=1$ and $a=1$, since the occurrence of the query predicate (`Testing`) matches with the department of the first tuple. The cloud computes the signbit (by placing $x=1$ and $a=1$) that results in 0, and subtracts it from 1 before multiplying by $r=3$ and $o=1$. Hence, the first column keeps the tuple id 3. The second column of the first row has 0, since $\mathit{signbit}(2-1)+\mathit{signbit}(1-2)=0+1=1$. Note that when processing the second row, the cloud finds the signbit of a equals to the value of the first column, while the second tuple does not have `Testing` department. The multiplication of the resultant of the signbit comparison by o makes the values of the first column 0, while the second column has 0 too. The cloud processes the third tuple like the first tuple. Here, the second column keeps the tuple id, since for the second column the current value of accumulated count a matches with the column number, while the first column stores 0, due to $1-(\mathit{signbit}(1-2)+\mathit{signbit}(2-1)=1-(1+0)$. The cloud processes the remaining tuples in a similar manner.

The second method. Now, we propose a method that takes at most three communication rounds, and in each round, each cloud sends $\sqrt{n}$ numbers to the user.

After performing the string-matching operation on the shares, we get a vector of n values that have either 1 or 0 of secret-share form. Thus, sending these n numbers are enough to reveal all the positions of a predicate in the database.

[13]The user either provides an upper bound on the number of tuples that can satisfy the query predicate or knows the occurrences of the query predicate by executing the count query.

TID (r)	Dept	SM result (o)	Count (a)	$x = 1$	$x = 2$
3	Testing	1	1	$r \times o[1-(signbit(x-1)+signbit(1-x))] = 3$	$r \times o[1-(signbit(x-1)+signbit(1-x))] = 0$
2	Security	0	1	$r \times o[1-(signbit(x-1)+signbit(1-x))] = 0$	$r \times o[1-(signbit(x-1)+signbit(1-x))] = 0$
5	Testing	1	2	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 5$
4	Design	0	2	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$
1	Design	0	2	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$
6	Design	0	2	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$	$r \times o[1-(signbit(x-2)+signbit(2-x))] = 0$
Tuple ids after adding values of the columns				3	5

Table III: Knowing tuple ids of employees working in testing department.

For example, in Table 1, if the user wants to know the row-ids of all Johns, the clouds only need to return the vector $\langle 1,1,0,0,0,0\rangle$ of secret-share form. The user can fetch the first and the second tuple obliviously be following the method given in §VI-B.[14] However, the scale of such a vector is equal to the number of rows in a database.

Now, we propose a novel compression method to reduce the communication cost. After doing string-matching operation on the cloud, each cloud obtains a vector X of length n in secret share form. Denoted by $X[i]$ the i-th entry of vector X. Then, this vector can be rewritten as a $\sqrt{n} \times \sqrt{n}$ matrix, say $\mathbf{M}_X$. The i-th entry of X now is denoted by $\mathbf{M}_X[a, b]$, where a is the quotient of i divided by $\sqrt{n}$ and $b = i \bmod \sqrt{n}$. The cloud performs an algorithm $Compress(\mathbf{M})$ (Algorithm 1), and the user performs $Eval(U, V)$ (Algorithm 2), where $\mathbf{M}$ is a $\sqrt{n} \times \sqrt{n}$ and U, V are vectors of length $\sqrt{n}$.

Algorithm 1: $Compress(\mathbf{M})$

Input: a $\sqrt{n} \times \sqrt{n}$ matrix $\mathbf{M}$.
Output: two vectors U and V of length $\sqrt{n}$.
for $i = 1$ *to* $\sqrt{n}$ **do** $U[i] = \sum_{j=1}^{\sqrt{n}} M[i, j]$, $V[i] = \sum_{j=1}^{\sqrt{n}} j * M[i, j]$.;
Return U, V.

Algorithm 2: $Eval(U, V)$

Input: two vectors U and V of length $\sqrt{n}$.
Output: matrix $\mathbf{M}$ which have 1 in the occurrences' positions and 0 otherwise.
Initialize $\mathbf{M}$ with 0;
for *i=1 to* $\sqrt{n}$ **do if** $U[i] = 1$ **then**
  $\mathbf{M}[V[i, i] = 1$
;
Return $\mathbf{M}$

Description: In our setting, all the data in clouds are in secret sharing form $S(*)$. However, the algorithm *Eval* cannot run over the secret sharing data. When the user obtains the vectors (secret shares) from the clouds, the user interpolate those secret shared values and obtain the vector and then run *Eval*. Based on the definition of *Eval*, one can check that if there is only one occurrence in the whole column, its position will be revealed. Otherwise, there maybe several possible positions. Additionally, if the clouds run the algorithm $Compress(\mathbf{M}^T)$, the user can also obtain more information about the occurrences' positions. However, after two rounds communications, the user will not always know the exact positions for all the occurrences. But two *Eval* can reveal rough positions for these occurrences, the third round communication is needed to reveal uncertain occurrences' positions. Moreover, the communication cost in such a round depends on the number of occurrences. As mentioned, we assume that the number of occurrences for each pattern is fewer than $\sqrt{n}$. In this case, we can recognize the matrix $\mathbf{M}_X$ be sparse and Round 1 and Round 2 will reveal enough information related to the uncertain positions.

[14]Here the DB owner has to add one more column, say `RowID` that has sequential numbers from 1 to n in the relation $S(R^1)$, which helps in finding the required tuples.

Full scheme. The whole scheme consists of three rounds. Each round requires clouds and user perform different operations.

Round 1: Clouds side: computes $Compress(\mathbf{M}_X)$ and sends $S(U_1), S(V_1)$ (secret-shared forms) to the user.
User side: interpolates all the shares, obtains U_1, V_1, and executes $Eval(U_1, V_1)$.

Round 2: Clouds side: computes $Compress(\mathbf{M}_X^T)$ and sends $S(U_2), S(V_2)$ (secret-shared forms) to the user.
User side: interpolates all the shares, obtains U_2, V_2 and executes $Eval(U_2, V_2)$.

Round 3: After two rounds communication, the user will obtain the explicit positions for the occurrences which satisfy *Eval*. Denoted by t the number of such occurrences. If t equals all the desired positions, Round 3 is not necessary. But if t is less than the desired positions, the user needs to get more information. Based on the result of Round 1 and 2, the user already know the possible positions for the result of occurrences. Therefore, during Round 3, all the clouds will send possible columns in the database to the user.

Example. We give a small example to explain the detailed scheme. For simplicity, we use plaintext in both the cloud and user side. Assume that there a database contains 100 rows and a pattern search lead to a search matrix as follows:

$$\mathbf{M}_X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \quad (1)$$

It is clear that there are seven occurrences in this database.

Round 1: Clouds compute $compress(\mathbf{M}_X)$, and set two vectors $U_1 = (0,3,0,3,0,1,0,0,0,0)$ and $V_1 = (0,12,0,12,0,6,0,0,0,0)$ to the user. The user checks U_1, V_1, only determines one occurrence's position. Then, the user writes the matrix as follows:

$$\mathbf{M} = \begin{bmatrix} 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad (2)$$

where X represent the possible positions of the rest of occurrences.

Round 2: Clouds compute $compress(\mathbf{M}_X^T)$, and set two vectors $U_2 = (0,2,0,2,0,3,0,0,0,0)$ and $V_2 = (0,6,0,6,0,12,0,0,0,0)$ to the user. The user checks U_2, V_2, and no exact position can be determined. But now the user can know the occurrences are in the row 2, 4, 6.

$$\mathbf{M} = \left[\begin{array}{c|c|c|c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & X & 0 & X & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]. \quad (3)$$

Now, the user checks the number of uncertain positions, which are reduced to 6. Since the user knows there are seven 1s in the matrix, the user replaces all X with 1 and recover the database.

Round 3: In this case, no third round of communication is needed. But sometimes if the user cannot recover all the occurrences' positions, the user will ask the clouds to send the data in all possible positions, which reveal the answer immediately.