OBSCURE: Information-Theoretic Oblivious and Verifiable Aggregation Queries*

Peeyush Gupta[1], Yin Li[2], Sharad Mehrotra[1], Nisha Panwar[1], Shantanu Sharma[1], and Sumaya Almanee[1]
[1]University of California, Irvine, USA. [2]Xinyang Normal University and Henan Key Lab of Analysis and Application of Educating Big Data, China.
Corresponding authors: sharad@ics.uci.edu, shantanu.sharma@uci.edu

ABSTRACT

Despite extensive research on cryptography, secure and efficient query processing over outsourced data remains an open challenge. We develop communication-efficient and information-theoretically secure algorithms for privacy-preserving aggregation queries using multi-party computation (MPC). Specifically, query processing techniques over secret-shared data outsourced by single or multiple database owners are developed. These algorithms allow a user to execute queries on the secret-shared database and also prevent the network and the (adversarial) clouds to learn the user's queries, results, or the database. We further develop (non-mandatory) privacy-preserving result verification algorithms that detect malicious behaviors, and experimentally validate the efficiency of our approach over large datasets, the size of which prior approaches to secret-sharing or MPC systems have not scaled to.

PVLDB Reference Format:
P. Gupta, Y. Li, S. Mehrotra, N. Panwar, S. Sharma, and S. Almanee. OBSCURE: Information-Theoretic Oblivious and Verifiable Aggregation Queries. *PVLDB*, 12(9): 1030-1043, 2019.
DOI: https://doi.org/10.14778/3329772.3329779

1. INTRODUCTION

Database-as-a-service (DaS) [33] allows authenticated users to execute their queries on an untrusted public cloud. Over the last two decades, several cryptographic techniques (*e.g.*, [31, 41, 29, 9, 40]) have been proposed to achieve secure and privacy-preserving computations in the DaS model. These techniques can be broadly classified based on cryptographic security into two categories:

Computationally secure techniques that assume the adversary lacks adequate computational capabilities to break the underlying cryptographic mechanism in polynomial time. Non-deterministic encryption [31], homomorphic encryption (HE) [29], order-preserving encryption (OPE) [9], and searchable-encryption [41] are examples of such techniques. HE mixed with oblivious-RAM (ORAM) offers the most computationally secure mechanisms.

Information-theoretically secure techniques that are unconditionally secure and independent of adversary's computational capabilities. Shamir's secret-sharing (SSS) [40] is a well-known information-theoretically secure protocol. In SSS, multiple (secure) shares of a dataset are kept at mutually suspicious servers, such that a single server cannot learn anything about the data. Secret-sharing-based techniques are secure under the assumption that a majority of the servers (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 discussed in [12].

The computationally/information-theoretically secure database techniques can also be broadly classified into two categories, based on the supported queries: (*i*) *Techniques that support selection/join*: Different cryptographic techniques are built for selection queries, *e.g.*, searchable encryption, deterministic/non-deterministic encryption, and OPE; and (*ii*) *Techniques that support aggregation*: Cryptographic techniques that exploit homomorphic mechanisms such as homomorphic encryption, SSS, or MPC techniques.

While both computationally and information-theoretically secure techniques have been studied extensively in the cryptographic domain, secure data management has focused disproportionately on computationally secure techniques (*e.g.*, OPE, homomorphic encryption, searchable-encryption, and bucketization [33]) resulting in systems such as CryptDB [38], Monomi [43], MariaDB [1], CorrectDB [11]). Some exceptions to the above include [25, 26, 45, 24] that have focused on secret-sharing.

Recently, both academia and industries have begun to explore information-theoretically secure techniques using MPC that efficiently supports OLAP tasks involving aggregation queries, while achieving higher security than computationally secure techniques.[1] For instance, commercial systems, such as Jana [10] by Galois, Pulsar [3] by Stealth Software, Sharemind [13] by Cybernetica, and products by companies such as Unbound Tech., Partisia, Secret Double Octopus, and SecretSkyDB Ltd. have explored MPC-based databases systems that offer strong security guarantees. Benefits of MPC-based methods in terms of both higher-level of security and relatively efficient support for aggregation queries have been extensively discussed in both scientific articles [39, 27, 21, 37] and popular media [4, 5, 6, 7].

*This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is partially supported by NSF grants 1527536 and 1545071. Y. Li' work is supported by National Natural Science Foundation of China (Grant no. 61402393, 61601396). The authors are thankful to the reviewers to help to improve the presentation.

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329779

[1]Computationally secure mechanisms may be vulnerable to computationally powerful adversaries, *e.g.*, Google, with sufficient computational capabilities, broke SHA-1 [2].

Much of the above work on MPC-based secure data management requires several servers to collaborate to answer queries. These collaborations require several rounds of communication among non-colluding servers. Instead, we explore secure data management based on SSS that does not require servers to collaborate to generate answers and can, hence, be implemented more efficiently. There is prior work on exploring secret-sharing for SQL processing [25, 26, 45, 24], but the developed techniques suffer from several drawbacks, *e.g.*, weak security guarantees such as leakage of access patterns, significant overhead of maintaining polynomials for generating shares at the database (DB) owner, no support for third-party query execution on the secret-shared outsourced database, etc. We discuss the limitations of existing secret-sharing-based data management techniques in details in §2.2.

Our contributions in this paper are twofold:

1. SSS-based algorithms (entitled 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, without revealing anything about data/query/results to an adversary.
2. An oblivious result verification algorithm for aggregation queries such that an adversary does not learn anything from the verification. OBSCURE's verification step is not mandatory. A querier may run verification occasionally to confirm the correctness of results.
3. A comprehensive experimental evaluation of OBSCURE on variety of queries that clearly highlight its scalability to moderate size datasets and its efficiency compared to both state-of-the-art MPC-based solutions, as well as, to the simple strategy of downloading encrypted data at the client, decrypting it, and running queries at the (trusted) client.

OBSCURE is designed to work both in situations when a single DB owner or when multiple DB owners outsource their data for others (queriers) to analyze. Example of a scenario with a single DB owner might be a hospital's patient DB outsourcing [8] to allow researchers to analyze the data. Example of a scenario with multiple DB owners might be utility owners outsourcing their smart meter data to enable others to analyze their utility consumptions as compared the average utility consumed by households in the region.

Outline of the paper. §2 provides an overview of secret-sharing techniques and related work. §3 and §4 provide the model, an adversary model, security properties, and data outsourcing model. §5 provides conjunctive/disjunctive count queries and their verification algorithm. §6 provides conjunctive/disjunctive sum queries and their verification algorithm. §7 provides an algorithm for fetching tuples having maximum values in some attributes with their verification. §8 provides an experimental evaluation.

Full version. Due to space limitations, we could not describe several technical details of OBSCURE, which can be found in the full version [32]. These include: detailed applications of OBSCURE, an approach for finding maximum over SSS databases outsourced by multiple DB owners, approaches for the minimum and top-k, an example of count query verification on secret-shares, an example of signbit computation on secret-shares, security proofs, a communication-efficient strategy for determining which tuples satisfied a query predicate, and range-queries evaluation.

2. BACKGROUND

Here, we provide an overview of secret-sharing with an example and compare our proposed approach with existing works.

2.1 Building Blocks

OBSCURE is based on SSS, string-matching operations over SSS, and order-preserving secret-sharing, discussed below.

Shamir's secret-sharing (SSS). In SSS [40], 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/servers. These servers cannot know the secret S until they collect $c' < c$ shares. In particular, the DB owner randomly selects a polynomial of degree c' with c' random coefficients, *i.e.*, $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{c'}x^{c'}$, where $f(x) \in \mathbb{F}_p[x]$, p is a prime number, $\mathbb{F}_p$ is a finite field of order p, $a_0 = S$, and $a_i \in \mathbb{N}(1 \leq i \leq c')$. The DB owner distributes the secret S into c shares by placing $x = 1, 2, \ldots, c$ into $f(x)$. The secret can be reconstructed based on any $c'+1$ shares using Lagrange interpolation [19]. Note that $c' \leq c$, where c is often taken to be larger than c' to tolerate malicious adversaries that may modify the value of their shares. For this paper, however, since we are not addressing the availability of data, we will consider c and c' to be identical.

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 server i, then the server 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. Accumulating-Automata (AA) [23] is a new string-matching technique on secret-shares that do not require servers to collaborate to do the operation, unlike MPC-techniques [15, 22, 35, 14, 13, 10]. Here, we explain AA to show string-matching operations 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} server, and c be the number of *non-communicating* servers. 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} server 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; otherwise. Note that when searching a pattern on the servers, AA uses *multiplication* of shares, as well as, additive property of SSS, which will be clear by the following example. If the user wishes to search a pattern of length l in only *one communication round*, while the DB owner and the user are using a polynomial of degree one, then due to multiplication of shares, the final degree of the polynomial will be $2l$, and solving such a polynomial will require at least $2l + 1$ shares.

Example. Assume that the domain of symbols has only three symbols, namely A, B, and C. Thus, A can be represented as $\langle 1, 0, 0\rangle$. Similarly, B and C can be represented as $\langle 0, 1, 0\rangle$ and $\langle 0, 0, 1\rangle$, respectively.

DB owner side. Suppose that the DB owner wants to outsource B to the (cloud) servers. Hence, the DB owner may represent B as its unary representation: $\langle 0, 1, 0\rangle$. If the DB owner outsources the vector $\langle 0, 1, 0\rangle$ to the servers, it will reveal the symbol. Thus, the DB owner uses any three polynomials of an identical degree, as shown in Table 1, to create three shares.

Table 1: Secret-shares of vector $\langle 0, 1, 0\rangle$, created by the DB owner.

Vector values	Polynomials	First shares	Second shares	Third shares
0	$0 + 5x$	5	10	15
1	$1 + 9x$	10	19	28
0	$0 + 2x$	2	4	6

User-side. Suppose that the user wishes to search for a symbol B. The user will first represent B as a unary vector, $\langle 0, 1, 0\rangle$, and then, create secret-shares of B, as shown in Table 2. Note that there is no need to ask the DB owner to send any polynomials to create shares or ask the DB owner to execute the search query.

Table 2: Secret-shares of vector $\langle 0, 1, 0\rangle$, created by the user.

Vector values	Polynomials	First shares	Second shares	Third shares
0	$0 + x$	1	2	3
1	$1 + 2x$	3	5	7
0	$0 + 4x$	4	8	12

Table 3: Multiplication of shares and addition of final shares by servers.

Computation on		
Server 1	Server 2	Server 3
$5 \times 1 = 5$	$10 \times 2 = 20$	$15 \times 3 = 45$
$10 \times 3 = 30$	$19 \times 5 = 95$	$28 \times 7 = 196$
$2 \times 4 = 8$	$4 \times 8 = 32$	$6 \times 12 = 72$
43	147	313

Server-side. Each server performs position-wise multiplication of the vectors that they have, adds all the multiplication resultants, and sends them to the user, as shown in Table 3. An important point to note here is that the server cannot deduce the keyword, as well as, the data by observing data/query/results.

User-side. After receiving the outputs ($\langle y_1 = 43, y_2 = 147, y_3 = 313\rangle$) from the three servers, the user executes Lagrange interpolation [19] to construct the secret answer, as follows:

$$\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} \times y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \times y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \times y_3$$
$$= \frac{(x-2)(x-3)}{(1-2)(1-3)} \times 43 + \frac{(x-1)(x-3)}{(2-1)(2-3)} \times 147 + \frac{(x-1)(x-2)}{(3-1)(3-2)} \times 313 = 1$$

The final answer is 1 that confirms that the secret-shares at the servers have B.

Note. In this paper, we use AA that utilizes *unary representation* as a building block. A recent paper Prio [20] also uses a unary representation; however, we use significantly fewer number of bits compared to Prio's unary representation. One can use Prio's unary representation too or use a different private string-matching technique over secret-shares.

Order-preserving secret-sharing (OP-SS). The concept of OP-SS was introduced in [25]. OP-SS 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 server. It is clear that finding records with maximum or minimum values using OP-SS is trivial. However, ordering revealed by OP-SS can leak more information about records. Consider, for instance, an employee relation, shown in Table 4 on page 5. For ease of exposition, we represent Table 4 in cleartext. In Table 4, the salary field can be stored using OP-SS. If we know (background knowledge) that employees in the *security* department earn 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 §7, by splitting the fields such as *salary* that can be stored using OP-SS, while storing other fields using SSS, we can benefit from the ordering supported by OP-SS without compromising security.

2.2 Comparison with Existing Work

Comparison with SSS databases. In 2006, Emekçi et al. [25] introduced the first work on SSS data for executing sum, maximum, and minimum queries. However, [25] uses a trusted-third-party to perform queries and is not secure, since it uses OP-SS to answer maximum/minimum queries. Another paper by Emekçi et al. [26] on OP-SS based aggregation queries requires the database (DB) owner to retain each polynomial, which was used to create database shares, resulting in the DB owner to store $n \times m$ polynomials, where n and m are the numbers of tuples and attributes in a relation. [26] is also not secure, since it reveals access-patterns (*i.e.*, the identity of tuples that satisfy a query) and using OP-SS.[2] Like [26], [45] proposed a similar approach and also suffers from similar disadvantages. [42] proposed SSS-based sum and average queries; however, they also require the DB owner to retain tuple-ids of qualifying tuples. [24] used a novel string-matching operation over the shares at the server, but it cannot perform general aggregations with selection over complex predicates. In short, all the SSS-based solutions for aggregation queries either overburden the DB owner (by storing enough data related to polynomials and fully participating in a query execution), are insecure due to OP-SS, reveal access-patterns, or support a very limited form of aggregation queries without any selection criteria.

In contrast, OBSCURE eliminates all such limitations. It provides a fully secure and efficient solution for implementing aggregation queries with selections. Our experimental results will show that OBSCURE scales to datasets with 6M tuples on TPC-H queries, the size of which prior secret-sharing and/or MPC-based techniques have never scaled to. The key to the efficient performance of OBSCURE still is exploiting OP-SS – while OP-SS, in itself is not secure (it is prone to background knowledge attacks, for instance). The way OBSCURE uses OP-SS, as will be clear in §4, it prevents such attacks by appropriately partitioning data, while still being able to exploit OP-SS for efficiency. In addition, to support aggregation with selections, OBSCURE exploits the string-matching techniques over shares developed in [23].

Furthermore, as we will see in experimental section (§8), OBSCURE scales to datasets with 6M tuples on TPC-H queries.

Comparison with MPC-techniques. OBSCURE also overcomes several limitations of existing MPC-based solutions. Recent work, Prio [20] supports a mechanism for confirming the maximum number, if the maximum number is known; however, Prio [20] does not provide any mechanism to compute the maximum/minimum. Also, Prio does not provide methods to execute conjunctive and disjunctive count/sum queries. Another recent work [14] deals with adding shares in an array under malicious servers and malicious users, using the properties of SSS and public-key settings. However, [14] is unable to execute a single dimensional, conjunctive, or disjunctive sum query. Note that (as per our assumption) though, [14] can tolerate malicious users, while OBSCURE is designed to only handle malicious servers, and it assumes users to be trustworthy.

Other works, *e.g.*, Sepia [15] and [22], perform *addition and less than* operations, and use many communication rounds. In contrast, OBSCURE uses minimal communication rounds between the user and each server, (when having enough shares). Specifically, count, sum, average, and their verification algorithms require at most two rounds between each server 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 server.

Comparison with MPC/SSS-based verification approaches. [35] and [42] developed verification approaches for secret-shared data. [35] considered verification process for MPC using a trusted-third-party verifier. While overburdening the DB owner by keeping metadata for each tuple, [42] provided metadata-based operation verification (*i.e.*, whether all the desired tuples are scanned or not) for only sum queries, unlike OBSCURE's result verification for all queries. OBSCURE verification methods neither involve the DB owner to verify the results nor require a trusted-third-party verifier.

3. PRELIMINARY

This section provides a description of entities, an adversarial model, and security properties for obliviously executing queries.

3.1 The Model

We assume the following three entities in our model.

[2]While [25, 26, 24, 10] have explored mechanisms to support selection and join operations over the secret-shared data, these techniques are not secure (*e.g.*, leak information from access-patterns), are inefficient (often requiring quadratic computations), and require transmitting entire dataset to users. SS can primarily be used to support OLAP style aggregation queries, which is our focus in this paper.

1. A set of $c > 2$ *non-communicating* servers. The servers do not exchange data with each other to compute any answer. The only possible data exchange of a server is with the user/querier or the database owner.
2. The *trusted* database (DB) owner, that creates c secret-shares of the data and transfers the i^{th} share to the i^{th} server. 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 server, as explained in §2.[3]
3. An (authenticated, authorized, and *trusted*) user/querier, who executes queries on the secret-shared data at the servers. The query is sent to servers. The user fetches the partial outputs from the servers and performs a simple operation (polynomial interpolation using Lagrange polynomials [19]) to obtain the secret-value.

3.2 Adversarial Model

We consider two adversarial models, in both of which the cloud servers (storing secret-shares) are not trustworthy. In the *honest but curious* model, the server correctly computes the assigned task without tampering with data or hiding answers. However, the server may exploit side information (*e.g.*, query execution, background knowledge, and output size) to gain as much information as possible about the stored data. Such a model is considered widely in many cryptographic algorithms and in widely used in DaS [17, 33, 44, 46]. We also consider a malicious adversary that could deviate from the algorithm and delete tuples from the relation. Users and database owners, in contrast, are assumed to be not malicious.

Only authenticated users can request query on servers. Further, we follow the restriction of the standard SSS that the adversary cannot collude with all (or possibly the majority of) the servers. Thus, the adversary cannot generate/insert/update shares at the majority of the servers. Also, the adversary cannot eavesdrop on a majority of communication channels between the user and the servers. This can be achieved by either encrypting the traffic between user and servers, or by using anonymous routing [30], in which case the adversary cannot gain knowledge of servers that store the secret-shares. Note that if the adversary could either collude with or successfully eavesdrop on the communication channels between the majority of servers and user, the secret-sharing technique will not apply.[4] The validity of the assumptions behind secret-sharing have been extensively discussed in prior work [39, 27, 21, 37]. The adversary can be aware of the public information, such as the actual number of tuples and number of attributes in a relation, which will not affect the security of the proposed scheme, though such leakage can be prevented by adding fake tuples and attributes.[5]

3.3 Security Properties

In the above-mentioned adversarial model, an adversary wishes to learn the (entire/partial) data and query predicates. Hence, a secure algorithm must prevent an adversary to learn the data (*i*) by just looking the cryptographically-secure data and deduce the frequency of each value (*i.e.*, frequency-count attacks), and (*ii*) when executing a query and deduce which tuples satisfy a query predicate (*i.e.*, access-pattern attacks) and how many tuples satisfy a query predicate (*i.e.*, output-size attacks). Thus, in order to prevent these attacks, our security definitions are identical to the standard security definition as in [16, 28, 18]. An algorithm is *privacy-preserving* if it maintains the privacy of the querier (*i.e.*, query privacy), the privacy of data from the servers, and performs identical operations, regardless of the user query.

Query/Querier's privacy requires that the user's query must be hidden from the server, the DB owner, and the communication channel. Also, the server 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, *e.g.*, all count queries are of the same type since they return almost an identical number of bits.

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

Privacy from the server requires that the stored input data, intermediate data during a computation, and output data are not revealed to the server, 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 server to prevent frequency analysis while data at rest. Recall that due to secret-shared relations (by following the approach given in §2.1), the server cannot learn the relations and frequency-analysis, and in addition, due to maintaining the query privacy, the server cannot learn the query and the output.

We, also, must ensure that the server's behavior must be identical for a given query, and the servers provide an identical answer to the same query, regardless of the users (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 servers 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 server if the real and ideal executions of the algorithm return an identical answer to the user.

Definition: Privacy from the server. *For any given secret-shared relation $S(R)$ at a server, 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 servers when computing a query. We follow the verification properties from [35], as follows: (*i*) the verification method cannot be refuted by the majority of the malicious servers, and (*ii*) the verification method should not leak any additional information.

3.4 OBSCURE Overview

Let us introduce OBSCURE at a high-level. OBSCURE allows single-dimensional and multi-dimensional conjunctive/disjunctive equality-based aggregation queries. 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[6] due to secret-shared data. Further, OBSCURE handles range-based

[3] The choice of underlying non-interactive and string-matching-based secret-sharing mechanisms do not change our proposed aggregation and verification algorithms.

[4] The DB owner/user can use anonymous routing to send their data to the servers, thereby preventing an adversary from determining which user is connecting to which server. If the adversary knows the majority of the communication channels/servers, then it can construct the secret-shared query, outputs to the query, and the database.

[5] 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.

[6] For the class of queries considered (viz. aggregation with selection), the main optimization in standard databases is to push selections down to determine whether an index-scan should be used or not. In SSS, an index scan cannot be used (at least not in any obvious way), since subsetting the data processed leads to revealing access-patterns, making the technique less secure. Thus, we avoid using an indexing structure.

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 servers using a secret-sharing mechanism that allows addition and multiplication (*e.g.*, [23]) at servers.

PHASE 2: *Query generation by the user.* The user generates a query, creates secret-shares of the query predicate, and sends them to the servers. For generating secret-shares of the query predicate, the user follows the strategies given in §5 (count query), §6 (sum queries), §7 (maximum/minimum), and §5.1,§6.1 (verification).

PHASE 3: *Query processing by the servers.* The servers process an input query in an oblivious manner such that neither the query nor the results satisfying the query are revealed to the adversary. Finally, the servers 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.

4. DATA OUTSOURCING

This section provides details on creating and outsourcing a database of secret-shared form. The DB owner wishes to outsource a relation R having attributes $A_1, A_2, \ldots, A_m$ and n tuples, and creates the following two relations R^1 and R^2:

• **Relation** R^1 that consists of all the attributes $A_1, A_2, \ldots, A_m$ along with two additional attributes, namely `TID` (tuple-id) and `Index`. As will become clear in §7, 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 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.[7]

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 randomly permuted. The reason of doing permutation is that the adversary cannot relate any tuple of both the secret-shared relations, which will be clear soon by the example below.

Note. The relation $S(R^1)$ will be used to answer count and sum queries, while it will be clear in §7 how the user can use the two relations $S(R^1)$ and $S(R^2)$ together to fetch a tuple having maximum/minimum/top-k/reverse-top-k value in an attribute.

Table 4: A relation: `Employee`.

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

Example. Consider the `Employee` relation (see Table 4). The DB owner creates R^1 = `Employee1` relation[8] (see Table 5a) with `TID` and `Index` attributes. Further, the DB owner creates R^2 = `Employee2` relation (see Table 5b) having three attributes `CTID`, `SSTID`, and `Salary`.

[7] If there are x attributes on which comparison operators is executed, the DB owner creates x relations, each with attributes `CTID`, `SSTID`, and one of the x attributes.

[8] 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.

Table 5: Two relations obtained from `Employee` relation.

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.

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 server (as specified in §2). 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: $S(R^1)_1, S(R^1)_2, \ldots, S(R^1)_c$, each having $m+2$ attributes. The notation $S(R^1)_k$ denotes the k^{th} secret-shared relation of R^1 at the server 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 server k.

Further, on the relation R^2, the DB owner creates c secret-shares of each value of `SSTID` using a secret-sharing mechanism that allows string-matching operations on the servers and each value of A_c using order-preserving secret-sharing [25, 34, 26]. 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} server. Also, by default, the attribute A_c have different shares in both the relations, due to using different secret-sharing mechanisms for different attributes. The DB owner outsources the relations $S(R^1)_i$ and $S(R^2)_i$ to the i^{th} server.

Note. Naveed et al. [36] showed that a cryptographically secured database that is also using order-preserving cryptographic technique (*e.g.*, OPE or OP-SS) may reveal the entire data when mixed with publicly known databases. Hence, in order to overcome such a vulnerability of order-preserving cryptographic techniques, we created two relations, and importantly, the above-mentioned representation, even though it uses OP-SS does not suffer from attacks based on background knowledge, as mentioned in §2. Of course, instead of using the two relations, the DB owner can outsource only a single relation without using OP-SS. In the case of a single relation, while we reduce the size of the outsourced dataset, we need to compare each pair of two shares, and it will result in increased communication cost and communication rounds, as shown in previous works [22, 15], which were developed to compare two shares.

5. COUNT QUERY AND VERIFICATION

This section presents 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. Consider a conjunc-

tive 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} server: `select count(*) from` $S(R^1)_j$ `where` $A_1 = S(v_1)_j \wedge A_2 = S(v_2)_j \wedge \ldots \wedge A_m = S(v_m)_j$. Each server 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 §2, $\otimes$ is a bit-wise multiplication and addition over a vector's elements, whose results will be 0 or 1 of secret-share form. Each server 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 server adds all those values. Since secret-sharing *allows the addition of two shares*, the sum of all n resultant shares provides the occurrences of tuples that satisfy the query predicate of secret-share form in the relation $S(R^1)$ at the j^{th} server. On receiving the values from the servers, the user performs Lagrange interpolation to get the final answer in cleartext.

Correctness. The occurrence of k^{th} tuple will only be included when the multiplication of m comparisons results in 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; thus, the k^{th} tuple will not be included. Thus, the correct occurrences over all tuples are included that satisfy the query's `where` clause.

Example. We explain the conjunctive count query method using the following query on the `Employee` relation (see Table 4): `select count(*) from Employee where Name = 'John' and Salary = 1000`. Table 6 shows the result of the private string-matching on the attribute `Name`, denoted by o_1, and on the attribute `Salary`, denoted by o_2. Finally, the last column shows the result of the query for each row and the final answer to the count query. Note that for the purpose of explanation, we use cleartext values; however, the server will perform all operations over secret-shares. For the first tuple, when the servers check the first value of `Name` attribute against the query predicate `John` and the first value of `Salary` attribute against the query predicate `1000`, the multiplication of both the results of string-matching becomes 1. For the second tuple, when the server checks the second value of `Name` and `Salary` attributes against the query predicate `John` and `1000`, respectively, the multiplication of both the results become 0. All the other tuples are processed in the same way.

Table 6: An execution of the conjunctive count query.

Name	o_1	Salary	o_2	$o_1 \times o_2$
John	1	1000	1	1
John	1	100000	0	0
Adam	0	5000	0	0
Eve	0	2000	0	0
Alice	0	1500	0	0
Mike	0	2000	0	0
				1

Disjunctive count query. Our disjunctive count query also scans the entire relation only once for checking multiple conditions of the query predicate. Consider a 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} server: `select count(*) from` $S(R^1)_j$ `where` $A_1 = S(v_1)_j \vee \ldots \vee A_m = S(v_m)_j$ The server 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 server 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:

$$temp_1^k = Result_1^k + Result_2^k - Result_1^k \times Result_2^k$$
$$temp_2^k = temp_1^k + Result_3^k - temp_1^k \times Result_3^k$$
$$\vdots$$
$$Output^k = temp_{m-1}^k + Result_m^k - temp_{m-1}^k \times Result_m^k$$

After performing the same operation on each tuple, finally, the server 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 is 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 server 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.

5.1 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, our objective is to verify that (*i*) all tuples of the databases are checked against the count query predicates, and (*ii*) all 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 server executes 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 server executes the functions f_1 and f_2 on n secret-shared values each (of two newly added attributes A_x and A_y, outsourced by the DB owner, described below). In the above equations, o_i is the output of 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 server 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 server has checked each tuple, respectively. The count query verification method 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 servers.

Server. Each server k executes the count query, as mentioned in §5, *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 server 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 server 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 server, which result in $Iresult$, Iop_1, and Iop_2. If the server 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 Table 4): `select count(*) from Employee where Name = 'John'`. Table 7 shows the result of the private string-matching, functions f_1 and f_2 at a server. Note that for the purpose of explanation, we use cleartext values; however, the server will perform all operations over secret-shares. For the first tuple, when the servers check 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 server performs each operation correctly.

Table 7: An execution of the count query verification.

`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

Correctness. Consider two cases: (*i*) all servers discard an entire identical tuple for processing, or (*ii*) all servers 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 servers 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 produce $Iresult = Iop_1$ at the user-side. Here, the user, however, finds the second condition $Iop_2 = n$ to be false.

Thus, the above verification method correctly verifies the count query result, always, under the assumption of SSS that an adversary cannot collude all (or the majority of) the servers, as given in §3.2.

6. SUM AND AVERAGE QUERIES

The sum and average queries are based on the search operation, as mentioned above in the case of count queries. This section, briefly, presents 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 a 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} server: `select sum(`A_ℓ`) from` $S(R^1)_j$ `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} server 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 the 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} server:

`select sum(`A_ℓ`) from` $S(R^1)_j$ `where` $A_1 = S(v_1)_j \vee A_2 = S(v_2)_j \vee \ldots \vee A_m = S(v_m)_j$

The server j executes the following computation:

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

$$Output = \sum_{k=1}^{k=n} A_\ell[S(a_k)]_j \times (((Result_1^k \texttt{ OR } Result_2^k) \texttt{ OR } Result_3^k) \ldots \texttt{ OR } Result_n^k)$$

The server 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 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 server to send the count and the sum of the desired values, and the user computes the average at their end.

Information leakage discussion. Sum queries work identically to count queries. Sum queries, like count queries, hide the facts which tuples are included in the sum operation, and the sum of the values.

6.1 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. `select sum(`A_ℓ`) from` R `where` $A_q = v$.

Here, our objective is to verify that (*i*) all tuples of the databases are checked against the sum query predicates, $A_q = v$, and (*ii*) only all 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 (§5.1). Further, the server 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 server executes the functions f_1 and f_2 on n values, described below. 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 server 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 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

Table 8: An execution of the sum query verification.

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

example, in Table 8, boldface numbers show these random numbers of the attribute A_x and A_y in cleartext.

Servers. The servers execute the above-mentioned sum query, *i.e.*, each server 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 server k adds all the resultant values of the attributes A_ℓ.

Verification stage. The server 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 server 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 output of the string-matching operations carried on the attribute A_q, say $\langle sum_q \rangle_k$,[9] 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 server, 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 server has correctly executed the sum query.

Example. We explain the above method using the following query on the `Employee` relation (refer to Table 4): `select sum(*) from Employee where Dept = 'Testing'`. Table 8 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 server. Note that for the purpose of explanation, we show the verification operation in cleartext; however, the server will perform all operations over secret-shares.

For the first tuple, when the server 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 server adds the salary of the first tuple to the first values of the attributes A_x and A_y with o_1. Then, the server multiplies the summation outputs by o_1.

For the second tuple, the servers perform 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 servers perform the same operations on the remaining tuples. Finally, the servers send 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. The occurrences of qualified tuples against a query predicates can be verified using the method given in §5.1. Consider two cases: (*i*) all servers discard an entire identical tuple for processing, or (*ii*) all servers 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 informs the user that an identical tuple is discarded by the server for any processing. In the second case, the user finds $Isum_{f1f2} - 2 \times Isum_q \neq Isum_\ell$, since an adversary cannot provide a wrong value of $Isum_q$, which 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 $Isum_{f1f2} - Isum_\ell = 2 \times Isum_q$, but an adversary cannot generate any share, as per the assumption of SSS that an adversary cannot produce a share, since it requires to collude with all (or the majority of) the servers, which is impossible due to the assumption of SSS, as mentioned in §3.2.

[9] If users are interested, they can also verify this result using the method given in §5.1.

7. MAXIMUM QUERY

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 * from Employee where Salary in (select max(Salary) from Employee)`

`QMax2.` `select * from Employee as E1 where E1.Dept = 'Testing' and Salary in (select max(salary) from Employee as E2 where E2.Dept = 'Testing')`[10]

Note that the string-matching secret-sharing algorithms (as explained in §2) 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 servers.

`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 maximum queries. Our idea is based on a combination of OP-SS [25, 34] and SSS [40, 23] 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 §3.1. In particular, according to our data model (§3.1), the attribute A_c will exist in the relations $S(R^1)_i$ and $S(R^2)_i$ at the server 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 Table 4), the maximum salary over all employees is unique.

7.1 Unconditional Maximum Query

Recall that by observing the shares of the attribute A_c of the relation $S(R^1)$, the server cannot find the maximum value of the attribute A_c. However, the server 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} server 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 server 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

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

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 server 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:
$$\sum_{k=1}^{k=n} A_p[S(a_k)]_i \times (\texttt{TID}[S(a_k)]_i \otimes S(t_k)_i)$$
Where p ($1 \leq p \leq m$) is the number of attributes in the relation R and TID is the tuple-id attribute of $S(R^1)_i$. The server 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. The server i adds all the values of each m attribute.
Correctness. The server 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 §3.1, 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 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 $S(R^2)$.

7.2 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 servers, as follows:

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

Round 2. The user interpolates the received indexes and sends the desired indexes in cleartext to the servers. Each server i finds the maximum value of the attribute A_c in the requested indexes 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 server 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 §7.1. 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 indexes of $S(R^1)$ 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 way as the method for answering unconditional maximum queries.

Information leakage discussion. In round 1, due to obliviously retrieving indexes of $S(R^1)$, the adversary cannot know which tuples satisfy the query predicate. In round 2, the user sends only the desired indexes in cleartext to quickly find the maximum salary. Note that by sending indexes, the adversary learns the number of tuples that satisfies the query predicate;[11] however, the adversary cannot learn which tuples of the relation $S(R^1)$ have those indexes. Due to OP-SS, the adversary also knows only the order of values of A_c attribute in the requested indexes. However, joining the tuple of $S(R^2)$, which has the maximum value in A_c attribute, 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.

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. Now, below, we provide a method to hide frequency-count information:
User-side. The user interpolates the received indexes (after round 1) and sends the desired indexes with some fake indexes, which do not satisfy the query predicate in the round 1, in cleartext to the servers. Let $x = r + f$ be the indexes that are transmitted to the servers, where r and f be the real and fake indexes, 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 indexes: The user arranges the x indexes in a $\sqrt{x} \times \sqrt{x}$ matrix, where all r real indexes appear before f fake indexes. 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 indexes. Now, the user asks the server 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}.
Server. For each group, g_j, except the group g_{mix}, each server 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 server i fetches all $\sqrt{x}$ tuples of the group g_{mix}. Then, the server i performs a join operation (based on the attribute TID and SSTID, as performed in the second step for answering unconditional maximum queries; see §7.1) 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, 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 servers and the user process more tuples than the method that reveals the frequency-count.

Obliviously finding the indexes. For finding the indexes, each server k executes the following operation: $\texttt{Index}[i]_k \times (A_p[i]_k \otimes S(v)_k)$, *i.e.*, the server executes string-matching operations on each

[11] 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.

Table 9: An execution of the tuple retrieval verification.

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
						$op_1 = 1733$	$op_2 = 500$

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 server 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 server 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 indexes.

7.3 Verification of Maximum Query

Verifying only the maximum value of the tuple is trivial, since $\langle S(value)\rangle_i$ of $S(R^2)_i$ is also a part of the attribute of A_c of $S(R^1)_i$, and servers send a joined output of the relations (see step 2 in §7.1). Thus, servers cannot alter the maximum value. However, servers can alter other attribute values of the tuple. Thus, we provide a method to verify the received tuple.

Verification of retrieved tuple. This method is an extension of the sum verification method (as given in §6.1). The server computes two functions, f_1 and f_2, in an oblivious manner, as follows:

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

i.e., the server executes the functions f_1 and f_2 on n values, described below. In the above equations, o_i is the output of the string-matching operation carried on the i^{th} value of the TID attribute, and $s_{i,j}$ be the i^{th} $(1 \leq i \leq n)$ value of the attribute j, where $1 \leq j \leq m$. The server sends the difference of the outputs of the functions f_1 and f_2 to the user. Particularly, the tuple verification method works as follows:

The DB owner. The DB owner adds one value to each of the attribute values of a tuple along with new attributes, say A_x and A_y.

Let A_1 be an attribute having only numbers. For A_1 attribute, the newly added i^{th} value in cleartext is same as the existing i^{th} value in A_1 attribute. Let A_2 be an attribute having English alphabets, say attribute Name in Employee relation in Table 4. The new value is the sum of the positions of each appeared alphabet in English letters; 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 supports string-matching at the server, as mentioned in §2.1, and the second value's shares are created using SSS.

The i^{th} values of the attributes A_x and A_y are two 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 of the i^{th} tuple. The values of the attributes A_x and A_y are secret-shared using SSS. E.g., in Table 9, numbers show newly added values to attributes Name′, Dept′, and random numbers (in boldface) of the attributes A_x and A_y in cleartext (a prime ($'$) symbol is used to distinguish these values from the original attribute values).

Servers. Each server k executes the method for tuple retrieval as given in step 2 in §7.1. Then, the server k executes functions f_1 and f_2, *i.e.*, adds all the m newly added values (one in each attribute) to x_i and y_i of the attributes A_x and A_y, respectively, and then, multiply the resultant of the string-matching operation carried on TID attribute of the relation $S(R^1)_k$. Finally, the server k sends the following two things to the user: (*i*) the tuple having the maximum value in the attribute A_c of the relation $S(R^1)_k$; and (*ii*) the difference of outputs of the functions f_1 and f_2, say $\langle diff_{f1f2}\rangle_k$.

Table 10: Exp. 1. Average time and size for shared data generation using *single-threaded implementation* at the DB owner.

Tuples	Time	Size (in GB)
1M	≈ 10 mins	$\lvert S(R^1)\rvert = 1.3, \lvert S(R^2)\rvert = 0.3$
6M	≈ 1.4 hours	$\lvert S(R^1)\rvert = 14, \lvert S(R^2)\rvert = 3$

User. After interpolation, the user obtains the desired tuple and a value, say $Idiff_{f1f2}$. Like the DB owner, the user generates a value for each of the attribute values of the received tuple (see the first step above for generating values), compares against $Isum_{f1f2}$, and if it finds equal, the server has correctly sent the tuple.

Example. Table 9 shows verification process for the first tuple-id of employee relation; see Table 4. Note that the values and computation are shown in the cleartext; however, the values are of secret-share form and the computation will be carried on shares at servers.

8. EXPERIMENTS

This section evaluates the scalability of OBSCURE and compares it against other SSS- and MPC-based systems. We used a 16GB RAM machine as a DB owner, as well as, a user that communicates with AWS servers. For our experiments, we used two types of AWS servers – a relatively weaker 32 GB, 2.5 GHz, Intel Xeon CPU (Exp 2, 5), and a powerful 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores to study impact of multi-threaded processing (Exp 3).

8.1 OBSCURE Evaluation

Secret-share (SS) dataset generation. We used four columns (Orderkey (OK), Partkey (PK), Linenumber (LN), and Suppkey(SK)) of LineItem table of TPC-H benchmark to generate 1M and 6M rows. To the best of our knowledge, this is the first such experiment of SSS-based approaches to such large datasets. We next explain the method followed to generate SS data for 1M rows. A similar method was used for generating SS data for 6M rows.

The four columns of LineItem table only contain numbers: OK: 1 to 300,000 (1,500,000 in 6M), PK: 1 to 40,000 (200,000 in 6M), LN: 1 to 7, and SK: 1 to 2000 (200,000 in 6M). The following steps are required to generate SS of the four columns in 1M rows:

1. The first step was to pad each number of each column with zeros. Thus, all numbers in a column contain identical digits, preventing an adversary to know the distribution of values. *E.g.*, after padding 1 of OK was 000,001. Similarly, values of PK and SK were padded. We did not pad LN values, since they took only one digit.
2. The second step was representing each digit into a set of ten numbers, as mentioned in §2.1, having only 0s or 1s. For example, 000,001 (one value of OK attribute) was converted into 60 numbers, having all zeros except at positions 1, 11, 21, 31, 41, and 52. Here, a group of the first ten numbers shows the first digit, *i.e.*, 0, a group of 11th to 20th number shows the second digit, *i.e.*, 0, and so on.[12] Similarly, each value of PK, SK, and LN was converted. We also added columns for TID, Index, count, sum, and maximum verification, and it resulted in the relation R^1. Further, we created another relation, R^2, with three attributes CTID, SSTID, and OK, as mentioned in §4.

[12] One may use binary representation for representing secret-shares, since it is compact as compared to unary representation. However, in binary representation, polynomial degree increases significantly, when we perform string-matching operations. For example, consider a decimal number, say n (= 400), having l_d (= 3) digits in decimal, and takes l_b (= 9) digits in binary (110010000). Here, representing 400 using unary and binary representations will take 30 and 9 numbers, respectively. However, when the user wishes to perform the minimum computation by interpolating only the desired answer, we need at least $2 \times l_d + 1$ and $2 \times l_b + 1$ servers for string-matching, when using unary and binary representation, respectively.

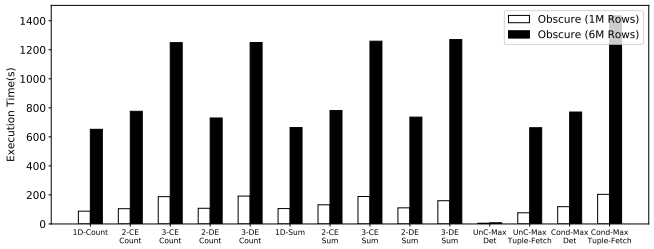


Figure 1: Exp 2. OBSCURE performance using **a single-threaded implementation on 32GB RAM**, 2.5GHz Intel Xeon CPU.

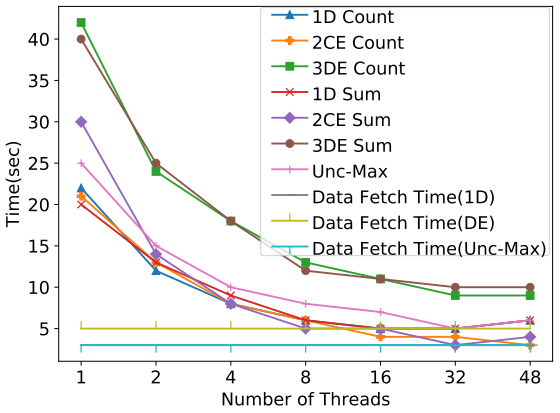
3. The third step was creating SS of these numbers. We selected a polynomial $f(x) = secret\_value + a_1 x$, where a_1 was selected randomly between 1 to 10M for each number, the modulus is chosen as 15,000,017, and x was varied from one to fifteen to obtain fifteen shares of each value. On R^2, we implemented OP-SS on OK attribute, and also generated fifteen shares of SSTID. Thus, we got $S(R^1)_i$ and $S(R^2)_i$, $1 \leq i \leq 15$. (Exp 5 will discuss in detail why are we generating fifteen shares.) For sum and tuple retrieval queries' time minimization, we add four more attributes corresponding to each of the four attributes in LineItem table. A value of each of the four attributes has only one secret-shared value, created using SSS (not after padding). But, one can also implement the same query on secret-shared values obtained after step 2.
4. Lastly, we placed i^{th} share of $S(R^1)$ and $S(R^2)$ to i^{th} AWS server.

Exp 1. Data generation time. Table 10 shows the time to generate secret-shared LineItem table of 1M and 6M rows, at the DB owner machine. Note that due to unary representation, the size of the data is large; but, the data generation time of OBSCURE is significantly less than an MPC system, which will be discussed in §8.2.

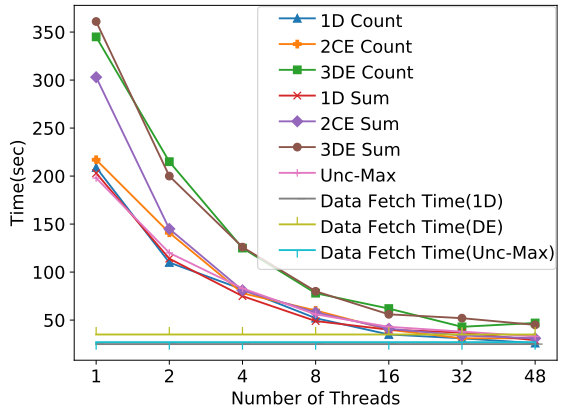
Exp 2. OBSCURE performance on a single-threaded implementation. This experiment explores OBSCURE on a relatively weaker single-threaded machine with 32GB. *We chose this machine since (as will be clear in §8.2) the used MPC system can only work on a local single-threaded machine. To be able to compare against that we also execute* OBSCURE *on 32GB AWS servers.* Note that single-threaded implementation of OBSCURE incurs time overheads, which are significantly reduced when using many threads on powerful servers; see next Exp 3. We executed count, sum, unconditional and conditional maximum queries on the LineItem table having 1M and 6M rows using fifteen shares; see Figure 1. Note that as the size of data increases, the time increases slightly more than linearly. This is due to the unary representation that requires 10 more numbers (for 6M rows table) to cover one new additional digit in all attribute values (except LN attribute). This increase results in additional multiplications during string-matching.

Count and sum queries. Figure 1 shows the time taken by one-dimensional (1D), two/three-dimensional conjunctive-equality (2CE/3CE), and two/three-dimensional disjunctive-equality (2DE/3DE) count and sum queries. CE queries were executed on OK and LN, and DE queries involved OK, PK, and LN attributes. Observe that as the number of predicates increases, the computation time also increases, due to an increasing number of multiplications. The time difference between computations on 1M and 6M rows is about 6-7.4%.

Maximum queries. Figure 1 shows that determining only the maximum value is efficient due to OP-SS, in case of unconditional maximum queries (UnC-Max-Det, `QMax1`, see §7). Time for determining the maximum value for conditional (Cond-Max-Det) query requires first executing a query similar to 1D/CE/DE sum query. We executed 1D conditional maximum query. The time is slightly more than executing 1D sum query, since Cond-Max-Det requires to know the tuple-ids that satisfy the condition in relation $S(R^1)$, and then, determining the maximum value from $S(R^2)$. Note that in both UnC-Max-Det and Cond-Max-Det, we achieve the maximum efficiently, due to OP-SS, (while also preventing background-knowledge-based attacks on OP-SS). The time difference between fetching a tuple having the maximum value from 1M and 6M data is about 5.5-6.6%.



(a) 1M rows.



(b) 6M rows.

Figure 2: Exp 3. Impact of parallelism, evaluated using **AWS servers of 144GB RAM, 3.0GHz Intel Xeon CPU, and 72cores.**

Exp 3. OBSCURE performance on multi-threaded implementation. In OBSCURE, the processing time at each server can be greatly reduced by parallelizing the computation. Since identical computations are executed on each row of the table, we can use multiple cores of CPU by writing a parallel program, which reduces the processing time. We wrote parallel programs (for 1D count/sum, 3DE count/sum, and unconditional maximum queries) that divide rows into blocks with one thread processing one block, and then, the intermediate results (generated by each thread) are reduced by the master thread to produce the final result. For this experiment having 15 shares, we used *AWS servers with 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores*, and varied the *degree of parallelism up to 48* (number of parallel threads). Increasing more threads did not provide speed-up, since the execution time reached close to the time spent in the sequential part of the program (Amdahl's law); furthermore, the execution time increases due to thread maintenance overheads. Figure 2 shows as the number of threads increases, the computation time decreases. Observe that the data fetch time from the database remains (almost) same and less than the processing time. Also, the computation time reduces significantly due to using many threads on powerful servers (Figure 2), than using a single thread on weaker servers (see Figure 1).

Exp 4. Impact of local processing at a resource-constrained user. To show the practicality of OBSCURE, we did an experiment, where a resource-constrained user downloads the entire encrypted

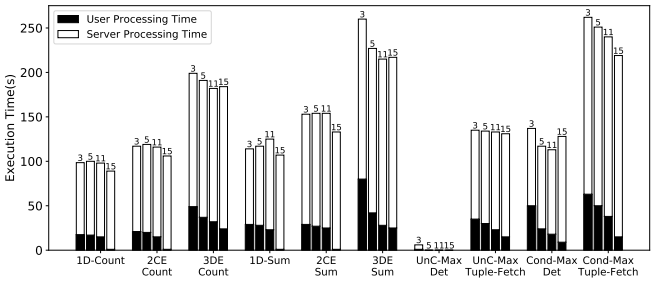


Figure 3: Exp 6. Impact of the number of shares.

data and executes the computation at their end after decrypting the data and loading into a database system. We restricted the user to have a machine with 1GB RAM and single core 1.35GHz CPU using docker, unlike multicore servers used in Exp 3, and executed the same queries that we executed in Exp 3. With this setup, decryption time at the user side was 54s and 259s for 1M and 6M rows, respectively. Further, loading decrypted data into a database system (MySQL) at the user-side took 20s and 120s for 1M and 6M rows, respectively. All queries used in Exp 2 were executed in 1-5s for both 1M and 6M rows. Note that the user computation time is significantly higher compared to the computation time of queries in Exp 3. For example, end-to-end 1D count query execution in Exp 3 over 6M secret-shared rows took 26s (see Figure 2b), while the same query took 385s when decrypting and loading the data into MySQL at the resource-constrained user.

Exp 5. Impact of the number of shares. Now, we discuss the impact of a different number of shares. For this experiment, we used 3, 5, 11, and 15 shares of the data. We show results of 1M rows. Figure 3 shows the computation time at the server and user, when having different shares, where black and white parts show the computation time at the user and server, respectively, and a bar shows the entire processing time. The results demonstrate two tradeoffs: between the number of shares and computation time at a user, and between the number of shares and the amount of data transferred from a server to a user. As the number of shares decreases, the computation time at the user increases; since the string-matching operation results in the polynomial degree to be doubled, and if servers do not have enough shares, they cannot compute the final answer and may require more than one round of communication with the user to compute the final SS aggregate value. Thus, the communication cost also increases.

From Figure 3, it is clear that as the number of shares increases, the computation time at user decreases and at the server increases. The total processing time also decreases for all queries, except 1D-Count, 2CE-Count, 1D-Sum, and 2CE-Sum. Due to space restrictions, we do not provide a detailed description of Figure 3, which is discussed in the full version of the paper [32].

Exp 6. Impact of communication cost. An interesting point was the impact of the communication cost. Since servers send data to the user over the network, it may affect the overall performance. As mentioned in Exp 4., using 3 servers, the communication cost increases as compared to 15 servers. For instance, in executing DE count/sum queries over PK, LN, and OK attributes took the highest amount of data transfer when using 3 servers. Since the number of digits of the three predicates was 12 in 1M rows and 14 in 6M rows, each server sends 12 files (each of size 7MB) in case of 1M rows and 14 files (each of size 48MB). Hence, the server to user communication was 84MB/server in case of 1M rows and 672MB/server in case of 6M rows. However, in case of 15 servers, the server to user communication was 7MB/server in case of 1M rows and 48MB/server in case of 6M rows. When using slow (100MB/s), medium (500MB/s), and fast (1GB/s) speed of data transmission, the data transmission time in case of 15 servers was negligible. However, in case of 6M, it took 7s, 1s, less than 1s per server, respectively, on slow, medium, and fast transmission speed.

Observe that the computation time at the server was at most 47s in any query on 6M rows (when using 72 core servers; Figure 2b) that was significantly more than the communication time between user and servers. Thus, *the communication time does not affect the servers's computation time, which was the bottleneck*.

Note. Experiments on result verification are omitted from here due to space restrictions and are given in [32].

8.2 Comparing with Other Works

The previous works on SSS-based techniques either did not report any experiments [25, 23] or scaled to only a very small dataset, or used techniques that, while efficient, were insecure [26, 45]. For instance, [26, 45] are both vulnerable to access-pattern attacks. Furthermore, these approaches achieve efficient query processing times (*e.g.*, 90 ms for aggregation queries on databases of size 150K) by executing queries on SS data identically to that on cleartext, which requires user sides to retain polynomials that were used to generate SS-data. Thus, as mentioned in §2.2, the DB owner keeps $n \times m$ polynomials, where n and m are the number of tuples and attributes in a relation, respectively.

MPC-based methods, *e.g.*, [15, 14, 10, 13], are secure; however, they also do not scale to large datasets due to high overhead of share creation and/or query execution. For example, MPC-based Sepia [15] used 65K values for only count operation without any condition with the help of three to nine servers, and recent paper [14] used only 500K values for count and sum of numbers. Note that Sepia [15] and [14] do not support conjunctive/disjunctive count/sum queries.

We evaluated one of the state-of-the-art industrial MPC-based systems, called system Z to get a better sense of its performance compared to OBSCURE, whose performance is given in Figure 1. We note that MPC systems, as mentioned in §1, are not available as open source (and often not even available for purchase, except in the context of a contract). We were able to gain access to System Z, due to our ongoing collaboration with the team under the anonymity understanding. We installed system Z (having three SS of LineItem) on a local machine, since it was not allowed to install it on AWS. Also, note that we cannot directly compare system Z and OBSCURE, since system Z uses a single machine to keep all three shares. Inserting 1M rows in system Z took 9 hours, while the size of SS data was 1GB. We executed the same queries using the system Z, which took the following time: 532s for 1D count, 808s for CE count, 1099s for DE count, 531s for 1D sum, 801s for CE sum, 1073s for DE sum, 2205s for UnC-Max-Tuple-Fetch, and 2304s for Cond-Max-Tuple-Fetch.

9. CONCLUSION

We proposed information-theoretically 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 proposed efficient result verification algorithms to protect against malicious adversarial cloud servers that deviate from the algorithm, due to software/hardware bugs. Our experimental results on 1M rows and 6M secret-shared rows using AWS servers show better performance as compared a simple strategy of downloading encrypted data, decrypting, and then, executing the query at a resource-constrained user. Further, we showed a tradeoff between the number of shares and performance. In the future, we plan to extend this work on GPU-based efficient join and nested queries.

10. REFERENCES

[1] MariaDB, available at:https://mariadb.com/.

[2] https://shattered.io/.

[3] Stealth Pulsar, available at:http://www.stealthsoftwareinc.com/.

[4] https://www.csoonline.com/article/3237685/identity-management/biometrics-and-blockchains-the-horcrux-protocol-part-3.html.

[5] https://bitcoinexchangeguide.com/binance-pays-6-cent-fee-for-moving-204-million-worth-of-ethereum-eth/.

[6] https://cryptoslate.com/thailands-democrat-party-holds-first-ever-election-vote-with-blockchain-technology/.

[7] https://blockonomi.com/coinbase-moves-5-billion-crypto/.

[8] R. Agrawal and C. M. Johnson. Securing electronic health records without impeding the flow of information. *I. J. Medical Informatics*, 76(5-6):471–479, 2007.

[9] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 563–574. ACM, 2004.

[10] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databases - real-world applications of secure multi-party computation. *IACR Cryptology ePrint Archive*, 2018:450, 2018.

[11] S. Bajaj and R. Sion. Correctdb: SQL engine with practical query authentication. *PVLDB*, 6(7):529–540, 2013.

[12] A. Beimel. Secret-sharing schemes: A survey. In *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, pages 11–46, 2011.

[13] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[14] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1175–1191. ACM, 2017.

[15] M. Burkhart, M. Strasser, D. Many, and X. A. Dimitropoulos. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 223–240. USENIX Association, 2010.

[16] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[17] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 639–648, 1996.

[18] C. Chu and W. Tzeng. Efficient *k*-out-of-*n* oblivious transfer schemes with adaptive and non-adaptive queries. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, pages 172–183, 2005.

[19] R. M. Corless and N. Fillion. A graduate introduction to numerical methods. *AMC*, 10:12, 2013.

[20] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282, 2017.

[21] R. Cramer, I. Damgård, and J. B. Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[22] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 285–304, 2006.

[23] S. Dolev, N. Gilboa, and X. Li. Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation: Extended abstract. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC@ASIACCS '15, Singapore, Republic of Singapore, April 14, 2015*, pages 21–29, 2015.

[24] S. Dolev, Y. Li, and S. Sharma. Private and secure secret shared MapReduce (extended abstract). In *Data and Applications Security and Privacy XXX - 30th Annual IFIP WG 11.3 Conference, DBSec 2016, Trento, Italy, July 18-20, 2016. Proceedings*, pages 151–160, 2016.

[25] F. Emekçi, D. Agrawal, A. El Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 27, 2006.

[26] F. Emekçi, A. Metwally, D. Agrawal, and A. El Abbadi. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.

[27] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. J. Weitzner. Practical accountability of secret processes. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 657–674, 2018.

[28] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 303–324, 2005.

[29] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[30] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[31] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[32] P. Gupta, Y. Li, S. Mehrotra, N. Panwar, S. Sharma, and S. Almanee. OBSCURE: Information-theoretic oblivious and verifiable aggregation queries. Technical Report, UCI, 2019. https://isg.ics.uci.edu/publications/.

[33] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD*

International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002, pages 216–227, 2002.

[34] M. A. Hadavi, E. Damiani, R. Jalili, S. Cimato, and Z. Ganjei. AS5: A secure searchable secret sharing scheme for privacy preserving database outsourcing. In *Data Privacy Management and Autonomous Spontaneous Security, 7th International Workshop, DPM 2012, and 5th International Workshop, SETOP 2012, Pisa, Italy, September 13-14, 2012. Revised Selected Papers*, pages 201–216, 2012.

[35] W. Jiang, C. Clifton, and M. Kantarcioglu. Transforming semi-honest protocols to ensure accountability. *Data Knowl. Eng.*, 65(1):57–74, 2008.

[36] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 644–655, 2015.

[37] C. Orlandi. Is multiparty computation any good in practice? In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pages 5848–5851, 2011.

[38] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: processing queries on an encrypted database. *Commun. ACM*, 55(9):103–111, 2012.

[39] A. Rajan, L. Qin, D. W. Archer, D. Boneh, T. Lepoint, and M. Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS 2018, Menlo Park and San Jose, CA, USA, June 20-22, 2018*, pages 49:1–49:4, 2018.

[40] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[41] D. X. Song, D. A. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[42] B. Thompson, S. Haber, W. G. Horne, T. Sander, and D. Yao. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *Privacy Enhancing Technologies, 9th International Symposium, PETS 2009, Seattle, WA, USA, August 5-7, 2009. Proceedings*, pages 185–201, 2009.

[43] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[44] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 253–262, 2010.

[45] T. Xiang, X. Li, F. Chen, S. Guo, and Y. Yang. Processing secure, verifiable and efficient SQL over outsourced database. *Inf. Sci.*, 348:163–178, 2016.

[46] S. Yu, C. Wang, K. Ren, and W. Lou. Attribute based data sharing with attribute revocation. In D. Feng, D. A. Basin, and P. Liu, editors, *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pages 261–270. ACM, 2010.