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, China.

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.

1. INTRODUCTION

Database-as-a-service (DaS) [34] allows authenticated users to execute their queries on an untrusted public cloud. Over the last two decades, several cryptographic techniques (*e.g.*, [33, 43, 31, 8, 41]) 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 [33], homomorphic encryption (HE) [31], order-preserving encryption (OPE) [8], and searchable-encryption [43] 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) [41] 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 [11].

The computationally or 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 [34]) resulting in systems such as CryptDB [39], Monomi [45], MariaDB [1], CorrectDB [10]). Some exceptions to the above include [27, 28, 47, 26] 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 [9] by Galois, Pulsar [3] by Stealth Software, Sharemind [12] 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 [40, 29, 22, 38] and popular media [4, 5, 6, 7].

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 [27, 28, 47, 26], 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 generat-

*This work of Y. Li is supported by National Natural Science Foundation of China (Grant no. 61402393, 61601396).

†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.

[1] Some of the computationally secure mechanisms are vulnerable to computationally powerful adversaries. For instance, Google, with sufficient computational capabilities, broke SHA-1 [2].

ing 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.

Applications. As mentioned, our proposed algorithms work for the case where a single or multiple DB owners outsource their data to the public servers. Here, we provide examples of each scenario.
Single database outsourcing. A hospital may outsource its patient database to an (untrusted) cloud vendor. Given sensitivity of the patient records, such data needs to be secured cryptographically. The hospital may still wish to execute analytical queries on the cloud over such data (e.g., number of influenza patients seen in the last month) for its own internal logistical planning.
DB outsourcing from multiple owners: Smart metering (or IoT sensors). Smart meters' data outsourcing is an example of multiple DB owners and a single querier. In smart meter settings, smart meter devices keep energy consumptions of a home at given time intervals and send the data to the servers [42]. This data contains behavioral information of the user; hence, a cryptographic technique should be used to make it secure before outsourcing. Users may execute queries on this secure database for monitoring and comparing their usage to that of others in the neighborhood. Executing such aggregate queries involve count, sum, and maximum operations in an oblivious manner at the server for preventing access to users' behavioral information. Our proposed algorithms prevent an adversarial server to learn the user's behaviors, when storing the database or executing a query.

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.

Appendix. In appendix, we provide the following: an approach for finding many tuples having the maximum value, an approach for finding maximum over SSS databases outsourced by multiple DB owners, approaches for the minimum, top-k, and reverse top-k, an outline for security proofs, and a communication-efficient strategy for knowing tuples that satisfied a query predicate.

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 (OP-SS). This section provides an overview of these existing techniques.

Shamir's secret-sharing (SSS). In SSS [41], 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 [20]. 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. Homomorphic Secret-Sharing (HSS) [14, 15] and Accumulating-Automata (AA) [24] are two new string-matching techniques on secret-shares that do not require servers to collaborate to do the operation, unlike MPC-techniques [16, 23, 36, 13, 12, 9]. Here, we explain AA to show how string-matching can be performed 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. Thus, if the user wants 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.

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

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

User-side. Suppose that the user wants 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.

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

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.

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

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

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 [20] 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 [21] 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 like HSS.

Order-preserving secret-sharing (OP-SS). The concept of OP-SS was introduced in [27]. 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 are trivial. However, ordering revealed by OP-SS can leak more information about records. Consider, for instance, an employee relation, given in Table 4 on page 5. For explanation purpose, 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, thus, can benefit from the ordering supported by OP-SS without compromising on security.

2's complement-based sigbit computation. [25] provided 2's complement-based sigbit computation. We will use signbit to find if two numbers are equal or not, as follows: $A \geq B$ if $signbit(A-B)=0$, and $A<B$ if $signbit(A-B)=1$. Let $A=[a_n,a_{n-1},\dots,a_1]$ be a n bit number and $B=[b_n,b_{n-1},\dots,b_1]$ be a n bit number. 2's complement subtraction converts $B-A$ into $B+\bar{A}+1$, where $\bar{A}+1$ is 2's complement representation of $-A$. We start from the least significant bit (LSB) and go through the rest of the bits. The method inverts a_i (by doing $1-a_i$, where $1\leq i\leq n$), calculates $\bar{a}_0+b_0+1$ and its carry bit. After finishing this on all the n bits, the most significant bit (MSB) keeps the signbit.

2.2 Comparison with Existing Work

Comparison with SSS databases. In 2006, Emekçi et al. [27] introduced the first work on SSS data for executing sum, maximum, and minimum queries. However, [27] 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. [28] 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. [28] 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 [28], [47] proposed a similar approach and also suffers from similar disadvantages. [44] proposed SSS-based sum and average queries; however, they also require the DB owner to retain tuple-ids of qualifying tuples. [26] 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 [24].

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 [21] supports a mechanism for confirming the maximum number, if the maximum number is known; however, Prio [21] 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 [13] deals with adding shares in an array under malicious servers and malicious users, using the properties of SSS and public-key settings. However, [13] is unable to execute a single dimensional, conjunctive, or disjunctive sum query. Note that (as per our assumption) though, [13] 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 [16] and [23], 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

[2] While [27, 28, 26, 9] 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.

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. [36] and [44] developed verification approaches for secret-shared data. [36] considered verification process for MPC using a trusted-third-party verifier. While overburdening the DB owner by keeping metadata for each tuple, [44] 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.

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 [20]) 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 may cryptographic algorithms and in widely used in DaS [18, 34, 46, 48]. 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 [32], 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 [40, 29, 22, 38]. 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 effect 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 [17, 30, 19]. 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. In addition, 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. For instance, 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.

Here, 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

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

[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.

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 [36], 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.

Algorithms' performance. We analyze our oblivious aggregation algorithms on the following parameters, which are stated in Table 6: (*i*) *Communication rounds*. The number of rounds that is required between the user and each server to obtain an answer to the query. (*ii*) *Scan cost at the server*. We measure scan cost at the server in terms of the number of the rounds that the server performs to *read the entire dataset*. (*iii*) *Computational cost at the user*. The number of values/tuples that the user interpolates to know the final output.

3.4 OBSCURE Overview

Let us introduce OBSCURE at a high-level. OBSCURE allows single-dimensional and multi-dimensional conjunctive/disjunctive equality 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 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.*, [24, 14, 15]) at the 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. The user can also verify these results by following the methods given in §5.1, §6.1, §7.3.

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 for 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.

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

Table 4: A relation: `Employee`.

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`.

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.

Table 5: Two relations obtained from `Employee` 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: $S(R^1)_1$, $S(R^1)_2$, ..., $S(R^1)_c$, 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 [27, 35, 28]. The secret-shares of the relation R^2 are denoted by $S(R^2)_i$ ($1 \leq i \leq c$). The

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

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

[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.

Algorithms	Query conditions	Scan rounds at a server	Comm. rounds	Interpolated values at user
Count §5	SD	1	1	1
	CQ	1	1	1
	DQ	1	1	1
Sum §6	SD	1	1	1
	CQ	1	1	1
	DQ	1	1	1
Unconditional max./min. (`SDBMax` §7.1)	One occurrence with tuple	1	1	m
Conditional maximum/minimum (`SDBMax` §7.2)	Finding maximum	1	2	$n+1$, $6\sqrt{n}+1$, or $\mathcal{T}+1$
	Tuple fetching	2	2	$n+m$, $6\sqrt{n}+m$, or $\mathcal{T}+m$
Maximum/Minimum (`MDBMax`) One /Multiple occurrences	Counting	$2n+1$	1	2
	Counting + tuple fetching	$2n+3$	3	$2\mathcal{T}+\ell m$
Maximum/Minimum (`SDBMax`) Many occurrences	Finding	2	2	1
	Tuple fetching	4	4	$2\mathcal{T}+\ell m$
Group-by		1	1	g
Top-k or reverse top-k	Unique occurrence	1 or k	2 or 1	$k \times m$

Notations. m: # attributes. n: # tuples. D: the database $n \times m$. 1D: Single dimensional equality query. CE: Conjunctive equality query. DE: Disjunctive equality query. $\mathcal{T}$: # tuple ids satisfying a query predicate. ℓ: # tuples having the maximum/minimum in the desired attribute. g: # groups. **Condition**: the above-mentioned rounds are given when we have $2l+1$ shares, where l is the maximum length of a keyword.

Table 6: Complexities of the algorithms.

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. [37] showed that a cryptographically secured database that is also using order-preserving cryptographic technique (*e.g.*, order-preserving encryption 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, as well as, communication rounds, as shown in previous works [23, 16], which were developed to compare two shares.

5. COUNT QUERY AND VERIFICATION

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

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

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` $A_1 = S(v_1)_j \land A_2 = S(v_2)_j \land \ldots \land A_m = S(v_m)_j$. Note that the single dimensional query will have only one condition. 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 [20] 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 above conjunctive count query method using the following query on the `Employee` relation (refer to Table 4): `select count(*) from Employee where Name = 'John' and Salary = '1000'`. Table 7 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 count answer for all the tuples. 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.

`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

Table 7: An execution of the conjunctive count query.

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

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 performs two functions, f_1 and f_2, as follows:

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

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

i.e., the 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 the string-matching operation carried on the i^{th} value of an attribute, say A_j, on which the user wants to execute the count query. The 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 verification method for a count query works as follows:

The DB owner. For enabling a count query result verification over any attribute, the DB owner adds two attributes, say A_x and A_y, having initialized with one, to the relation R^1. The values of the attributes A_x and A_y are also outsourced of SSS form (not unary representations) to the 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. In Appendix A, we provide an example of count query verification over secret-shares. However, here, 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 8 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.

`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

Table 8: An execution of the count query verification.

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 conjunctive/disjunctive count queries. In this section, we briefly present sum and average queries on a secret-shared database outsourced by single or multiple DB owners. Then, we develop a result verification approach for sum queries.

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

In the secret-sharing setting, the user transforms the above query into the following query at the j^{th} 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 example, in Table 9, 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, then it implies that 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 9 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 will inform the user that an identical tuple is discarded by the server for any processing. In the second case, the user

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

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

Table 9: An execution of the sum query verification.

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 all (or the majority of) the servers, which is impossible due to the assumption of SSS, as mentioned in §3.2.

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 [27, 35, 14] and SSS [41, 24, 14, 15] 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 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. Finally, 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).

[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.

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 fasten the lookup of 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 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.[12]

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
						$op_1 = 1733$	$op_2 = 500$

Table 10: An execution of the tuple retrieval verification.

7.3 Verification of Maximum Query

This section provides a method to verify the tuple having maximum value in an attribute, A_c. Note that 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

[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.

[12] The servers can also check conjunctive and/or disjunctive conditions, like one-dimensional condition (see §5 to recall the method of evaluating conjunctive and/or disjunctive conditions). Here, the server multiplies the i^{th} resultant of conjunctive and/or disjunctive conditions matching by the i^{th} value of `Index` attribute of the relation $S(R^1)$, and then, sends all the n values of the attribute `Index` to the user.

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 10, 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 $\langle \mathit{diff}_{f1f2} \rangle_k$.

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, then it implies that the server has correctly sent the tuple.

Example. Table 10 shows the 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, 6), and a powerful 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores to study impact of multi-threaded processing (Exp 3, 8).

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. Hence, all numbers in a column contain identical digits to prevent an adversary to know the distribution of values. For example, 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 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.[13] 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.
3. The third step was creating SS of these numbers. We selected a polynomial $f(x) = \mathit{secret\_value} + a_1x$, 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 11 shows the time to generate secret-shared LineItem table of size 1M and 6M rows, at the DB owner machine. Note that due to unary representation, the size of the data is large; however, the data generation time of OBSCURE is significantly less than an MPC system, which will be discussed in §8.2.

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

Table 11: Exp. 1. Average time and size for shared data generation using single-threaded implementation.

Exp 2. OBSCURE performance on a single-threaded implementation. In this experiment, we explored OBSCURE on a relatively weaker single-threaded machine with 32GB. *We chose this machine since (as will be clear §8.2) the MPC system, we used, 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, and group-by 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 the 6M row database) to cover one new additional digit in all attribute values (except LN attribute). This increase results in additional multiplications during string-matching.

[13] 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.

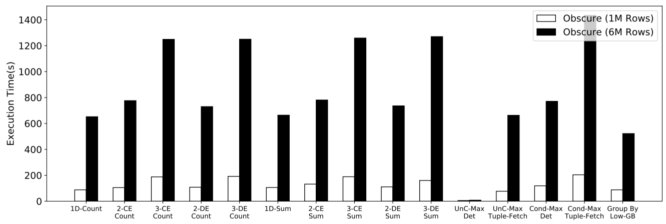


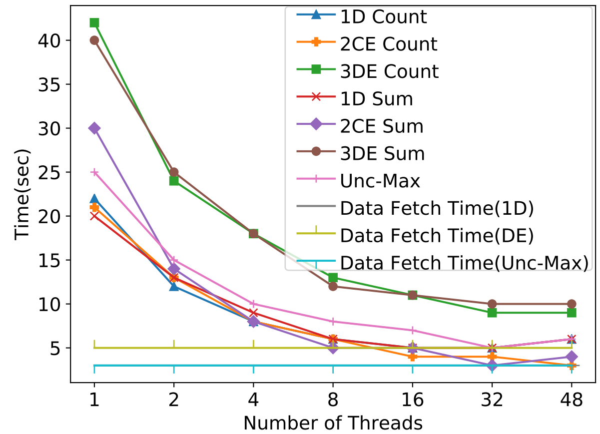
Figure 1: Exp 2. OBSCURE performance using **a single-threaded implementation on 32GB RAM**, 2.5GHz Intel Xeon CPU.

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%.

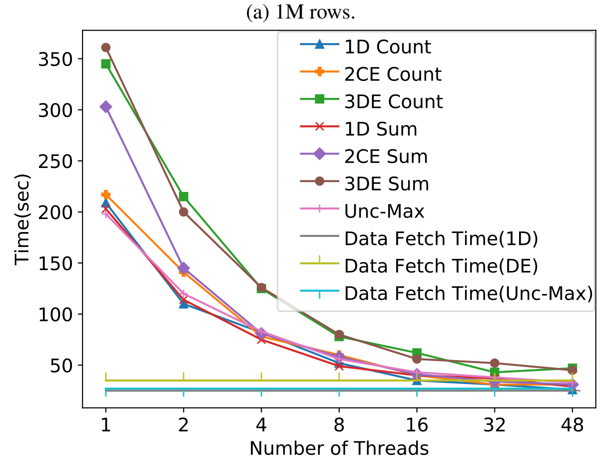
Maximum queries. Figure 1 shows that determining only the maximum value is efficient due to OP-SS, in case of unconditional maximum queries (UnC-Max-Det, `QMax1`, see §7). Time for determining the maximum value for conditional (Cond-Max-Det) query requires first executing a query similar to 1D/CE/DE 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%.

Group-by queries. Though we have not formally discussed a group-by query, the reason to include the group-by query in the experiment is that, in short, a group-by query works in a similar manner to 1D count/sum query. Details about group-by queries are given in Appendix C. Figure 1 shows the time taken by a group-by query when the number of groups was seven (Low-GB, due to LN attribute that has seven values). We executed a query to count the number of OK values corresponding to each LN value.

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. Also, observe that the data fetch time from the database remains (almost) same and less than the processing time. Further, 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).



(a) 1M rows.



(b) 6M rows.

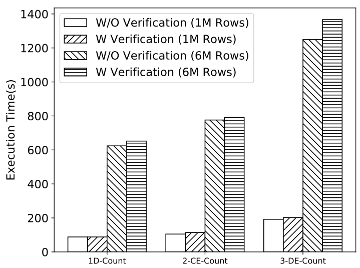
Figure 2: Exp 3. Impact of parallelism, evaluated using **AWS servers with 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores.**

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 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.

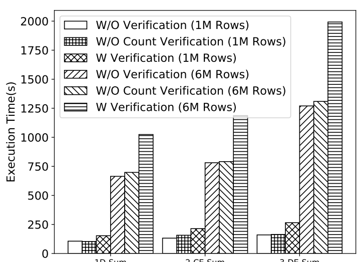
Exp 5. Overheads of result verification. This experiment finds the overheads of the result verification approaches. Figure 3a shows that count result verification steps do not incur a significant cost at the servers, since executing result verification requires only two more multiplications and modulo on each row's A_x and A_y values (see §5.1). However, in case of a sum query, the cost increases, due to first verifying count query results, and then, sum query results. If one drops count query result verification, the cost decreases significantly; see Figure 3b. Figure 3c shows the time comparison between fetching a tuple having the maximum value in an attribute and verifying that tuple. Here, in case of UnC-Max-Tuple-Fetch, this step does not involve any condition checking. However, in case of Cond-Max-Tuple-Fetch, we need to first apply count query verification method to verify that query predicate(s) are evaluated correctly. As mentioned previously, we are evaluating conditional maximum query for 1D predicate; hence, this step increases the time of verification by 304 and 790 seconds (s), in case of 1M and 6M rows, respectively.

Exp 6. Impact of number of shares. In this experiment, we study the impact of number of shares on the performance of OBSCURE. For this experiment, we used four different setups with data, secret shared between 3, 5, 11, and 15 servers. Due to space restrictions, we show results for 1M rows only. Figure 4 shows computation time at the server and user side, with different number of shares.

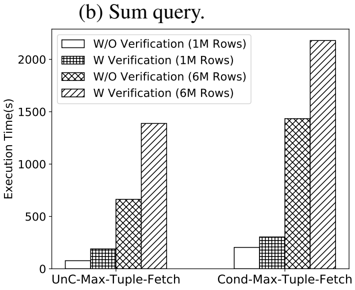
The results demonstrate two tradeoffs, first between the number of shares and computation time at the user, and second between the number of shares and the amount of data transferred from each server to the user. As the number of shares decreases, the computation time at the user increases; since the string-matching operation results in the degree of polynomials 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 SS aggregate value. Thus, the communication cost also increases with decreasing number of shares.



(a) Count query.



(b) Sum query.



(c) Maximum tuple fetch query.

Figure 3: Result verification.

From Figure 4, it is clear that as the number of shares increases, the computation time at the user decreases and at the server increases, while the overall query execution time decreases, generally. Below we discuss the processing of each query under different number of shares.

1D-count query. Consider a 1D-count on OK attribute (see Figure 4). Each OK value needs six digits, which we denote as: $\langle d_1, d_2, d_3, d_4, d_5, d_6 \rangle$. In order to evaluate a query predicate over OK attribute in a 1D-count query, using one round of communication between the user and the servers, we need at most thirteen servers/shares, as mentioned in §2.

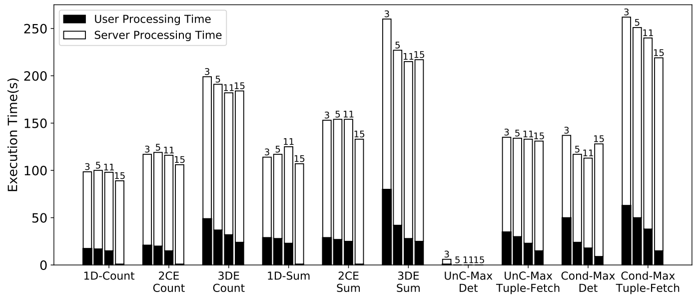


Figure 4: Impact of the number of shares, using a single threaded implementation on 32GB RAM, 2.5GHz Intel Xeon CPU.

When using three servers, the computation time at each server and the user was 81s and 17s, respectively. Here, the servers can only compare individual digits of each OK value against the query predicate; they can not evaluate the entire query predicate by comparing the entire OK value. Thus, the server sends partial results corresponding to each digit of OK value to the user. For each tuple in the result, the user, then, interpolates string-matching resultant of each digit, multiplies them, and finally, adds 1M values, resulting in an answer to the count query.

In the case of five shares, each server checks two digits of each OK value against the query predicate, *i.e.*, the server checks $\langle d_1, d_2 \rangle$, $\langle d_3, d_4 \rangle$, and $\langle d_5, d_6 \rangle$, and sends the partial results to the user. Note that checking two digits of each OK value requires more multiplication and modulus operations than using three shares, and thus, the server computation time increases to 83s. Here, the user receives smaller set of partial results, and thus, the user's interpolation task reduces. Note that the total time when using five shares is higher than three shares, since the server performs more computations. In case of eleven shares, the computation time at the server is higher as compared to three and five shares, since the server is able to check five digits ($\langle d_1, d_2, d_3, d_4, d_5 \rangle$) of the query predicate. In the case of fifteen servers, each server checks the entire predicate, and hence more computation time is required at the server, due to more multiplication and modulo operation. However, in the case of fifteen shares, the user pays only for interpolating one value, which is the answer to the count query.

2CE-count. Now, consider a 2CE-count query on OK and LN attributes, where the number of digits in OK and LN were 6 and 1 respectively. In order to execute a 2CE-count query, we need at most fifteen shares. A 2CE-count query execution time follows a trend similar to 1D-count query (see Figure 4) except that there are more number of digits involved. Hence, we skip details here.

3DE-count. 3DE-count queries were executed on OK, PK, and LN attributes, which have in total 12 digits, hence, we need at least 25 shares to compute the answer of a 3DE-count query in one round. Since we use at most fifteen shares, the servers sends the partial results of string-matching, as 1D-count query. The user interpolates them and obtains the answer at their end. This query follows a similar trend like 1D-count query; hence, we omit details here.

1D-sum. Sum queries behave differently than count queries, with increasing number of shares. 1D-sum queries include query predicate on OK attribute, wherein each value has six digits, which we represent as $\langle d_1, d_2, d_3, d_4, d_5, d_6 \rangle$. Generally, in a sum query, if the server does not have enough shares, they need to communicate with the user, who reduces the degree of polynomial of searching predicate attribute, and then, the user again sends the shares of string-matching resultant to the server. Afterwards, the server performs sum operations by multiplying i^{th} secret-shared result with the i^{th} value of desired attribute on which the user is executing a sum query, and then, the server adds all the values.

As we increase the number of servers from three to fifteen, the server computation time increases, due to more computations, like count 1D-count queries, and the user time decreases. Also, here, the total time increases when going to three to five shares, like 1D-count query. However, note that the time when using eleven shares, in case of 1D-sum query is higher than five shares, unlike 1D-count query. In case of eleven shares, the server checks $\langle d_1, d_2, d_3, d_4, d_5 \rangle$ and sends their output to reduce the degree of polynomial. However, the server does not send the output of string-matching operation over $\langle d_6 \rangle$ digit. Now, the user creates five shares of each value after reducing the degree. On receiving new shares, the server multiplies the i^{th} new share to the output of $\langle d_6 \rangle$ digit string-matching, whose resultant is multiplied by i^{th} share of the attribute on which the sum operation is carried out. Note that after this multiplication, the degree of polynomial is four; thus, the user sent five shares to recover the secret value. Note that while user time is almost same when using five or eleven shares, the server time is higher in case of eleven shares, since the server is matching almost the entire query predicate, except the last digit. Thus, the server time in the case of eleven shares is higher than five shares. In case of fifteen shares, the user time is minimum, since servers sends final answer to the sum query, while server time is maximum.

2CE-sum query. Now, consider a 2CE-sum query on OK and LN attributes, where the number of digits in OK and LN were six and one, respectively. In order to execute this 2CE-sum query, we need at most fifteen shares. A 2CE-sum query execution time follows a similar trend like of a 1D-sum query (see Figure 4). Hence, we omit details here.

3DE-sum query. 3DE-sum query is also executed similar to a 1D-sum query; hence, we omit details here.

UnC-Max-Tuple-Fetch query. While retrieving a tuple having the maximum value for an attribute, say A_c, the server joins two relations $S(R^1)$ and $S(R^2)$, based on `TID`, as mentioned in §7.1. For both 1M and 6M tuples dataset, each `TID` consists of seven digits, which we represent as $\langle d_1, d_2, d_3, d_4, d_5, d_5, d_6, d_7 \rangle$. When increasing number of shares from three to fifteen, the server time increases and the user time decreases, similar to all the above mentioned queries. Moreover, in this case, the total time of computation is decreasing. For this the reason is as follows:

During the join operation, the degree of polynomial used to create shares of `TID` values increases. In order to execute string-matching over `TID` values and tuple retrieval in one round, we need at least fifteen and sixteen shares, respectively. Thus, in our setting, the user needs to reduce the degree of string-matching resultant and re-generate shares of this to fetch the tuple, regardless of three, five, eleven, or fifteen shares.

In case of three shares, the server compares only each digit and sends partial results to the user. After interpolating the partial results, which consists of zeros for every tuple except for one, for each tuple, the user creates three secret-shares of this vector and sends to the servers to retrieve the desired tuple. This operation require interpolating seven shares, and then, generating three new shares.

In the case of five shares, the server compares and sends partial results of string-matching over $\langle d_1, d_2 \rangle$, $\langle d_3, d_4 \rangle$, $\langle d_5, d_6 \rangle$ to the user. Note that the user interpolates three shares and generates five new shares. Hence, the user time decreases as compared to the case of three shares.

In case of eleven shares, each server sends partial results of string-matching over $\langle d_1, d_2, d_3, d_4, d_5 \rangle$ its share, and the user generates five new shares. Thus, the user time again decreases in this case. In case of fifteen servers, each server checks the entire `TID` value and sends partial results for degree reduction. After interpolating the values, the user generates three new secret-shared files. Thus, the user time again decreases in this case as compared to eleven servers. In addition, as the servers checks more number of digits in the `TID` value, their time increases.

Cond-Max-Det. As mentioned in §7.2, finding the maximum value for conditional query requires at least two rounds of communication, when having enough shares. For this query, we set query predicate on OK attribute, which has six digits in every value. Hence, checking the query predicate on OK values in only one communication round, requires at least thirteen shares.

In case of less number of shares (*e.g.*, three, five, or eleven), the server first checks partial query predicates and sends results to the user for degree reduction, like 1D-count query. The user decreases the degree of string-matching resultant and sends new shares, where the new i^{th} share gets multiplied by i^{th} value of `Index` attribute to know the tuple ids. Finally, the server sends the resultant to the user. After interpolation, the user knows the tuple ids that satisfy the query predicate. Hence, in the case of three, five, or eleven shares, the user executes the interpolation operation two times, while in case of fifteen shares, the user executes interpolation operation only one time. Hence, the user computation time reduced when increasing the number of shares. After knowing tuple ids, the user asks the server to find the maximum value in the given tuple ids using the relation $S(R^2)$, and this operation takes same time regardless of number of shares. While increasing the number of shares, the server computation time increases, as it happened in all above-mentioned queries.

Cond-Max-Tuple-Fetch. Fetching a tuple having maximum value according to a conditional query requires two rounds, as stated in §7.2. The first round at the server is identical to Cond-Max-Det queries. However, in the second round, the server joins $S(R^1)$ with one of the tuple of $S(R^2)$ based on `TID` attribute of $S(R^1)$. In this query, we set a condition on OK attribute, which requires six digits to represent a value. Hence, we need at least $2 \times 6 + 1 = 13$ shares to know the tuple ids in one communication round. Further, to get the desired tuple, based on join over `TID`, that has seven digits, we need at least sixteen shares (fifteen shares for string-matching operations ($2 \times 7 + 1 = 15$) and one more share for reconstructing the tuple values).

In the first round, the user interpolates at least twice in case of three, five, and eleven shares, and at least once in case of fifteen shares to know the tuple ids. Further, we use at most fifteen shares in our experiments; hence, the user needs to reduce the degree at least once, of string matching resultant in the second round to get the desired tuple. Thus, user computation time decreases as the number of shares increases. Further, the server time decreases, as we increase the number of shares, similar to other queries.

Exp 7. 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 least 40s 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*.

Exp 8. Range queries. We evaluated range queries for 1D-count and 1D-sum operations. Given a range query involving k continuous values, we converted it into k 1D-count/sum queries (one per value in the range). However, this may require scanning the secret-shared relation k-times at the server. In order to reduce the number of scans, we processed (as per 1D-count or sum query) all the k-values in the range on each tuple, before processing the next tuple. As a result, we got k values (as per the 1D-count or sum query) after processing the entire relation. Finally, the server adds all k values and sends them to the user. We implemented a range query involving 1D-count/sum operations, using 48 threads on AWS servers with 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores. Figure 5 shows that as the length of range increases the computation time also increases. In Appendix F, we provide a bucketization-based approach to reduce the computation time while increasing the range values.

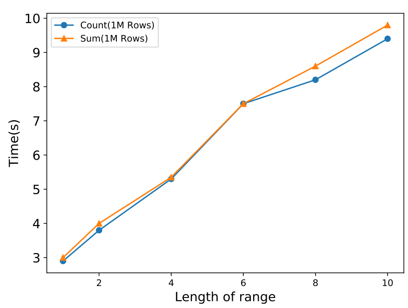


Figure 5: Impact of executing range queries.

8.2 Comparing with Other Works

The previous works on SSS-based techniques either did not report any experiments [27, 24] or scaled to only a very small dataset, or used techniques that, while efficient, were insecure [28, 47]. For instance, [28, 47] 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, which 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 rows and columns in a database, respectively.

MPC-based methods, *e.g.*, [16, 13, 9, 12], are secure, they also do not scale to large datasets due to high overhead of share creation and/or query execution. For example, MPC-based Sepia [16] used 65K values for only count operation without any condition with the help of three to nine servers, and recent Bonawitz et al. [13] (appeared in CCS 2017) used only 500K values for count and sum of the numbers. Note that Sepia [16] and Bonawitz et al. [13] 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 the MPC systems, as mentioned in §1, are not available as either 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 the 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, 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù. Homomorphic secret sharing: Optimizations and applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2105–2122, 2017.

[15] E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro. Foundations of homomorphic secret sharing. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 21:1–21:21, 2018.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] S. Dolev and Y. Li. Secret shared random access machine. In *Algorithmic Aspects of Cloud Computing - First International Workshop, ALGOCLOUD 2015, Patras, Greece, September 14-15, 2015. Revised Selected Papers*, volume 9511 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

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

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] L. V. Silva, R. Marinho, J. L. Vivas, and A. Brito. Security and privacy preserving data aggregation in cloud computing. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1732–1738. ACM, 2017.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

APPENDIX

A. COUNT QUERY VERIFICATION OVER SECRET-SHARED VALUES

This section shows an example for count query verification over a secret-shared relation.

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

DB owner side. Suppose that the DB owner wants to outsource three rows having A, B, A, respectively. The DB owner adds two attributes, A_x and A_y, initialized with one, to the relation; see 12.

Values	A_x	A_y
A	1	0
B	1	0
A	1	0

Table 12: Non-secret-shared relation at the DB owner.

The DB owner uses any polynomials of an identical degree, as shown in Table 13, to create four shares. Further, the i^{th} share is placed to the i^{th} server.

Values	Vector values	Polynomials	First shares	Second shares	Third shares	Fourth shares
A	1	$x+1$	2	3	4	5
	0	$3x+0$	3	6	9	10
B	0	$4x+0$	4	8	12	16
	1	$2x+1$	3	5	7	9
A	1	$5x+1$	6	11	16	21
	0	$4x+0$	4	8	12	16
A_x	1	$x+1$	2	3	4	5
	1	$2x+1$	3	5	7	9
	1	$4x+1$	5	9	13	17
A_y	1	$3x+1$	4	7	10	13
	1	$5x+1$	6	11	16	21
	1	$2x+1$	3	5	7	9

Table 13: Secret-shares of a relation shown in Table 12.

User-side. Suppose that the user wants to search for a symbol B. The user will first represent B as a unary vector, $\langle 0, 1\rangle$, and then, create secret-shares of B, as shown in Table 14.

Vector values	Polynomials	First shares	Second shares	Third shares	Fourth shares
0	$2x+0$	2	4	6	8
1	$x+1$	2	3	4	5

Table 14: Secret-shares of a vector $\langle 0, 1\rangle$, created by the user/querier.

Server-side. Each server executes the count query, as mentioned in §5, and the functions f_1 and f_2.

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

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

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). Each server i ($1 \leq i \leq 4$) sends the following three things: (*i*) the result of the count query $\langle result\rangle_i$, (*ii*) the outputs of the function f_1: $\langle op_1\rangle_i$, and (*iii*) the sum of outputs of the function f_1 and f_2: $\langle op_2\rangle_i$, to the user. Tables 15 - 18 show the working of servers over secret-shares.

User-side. The user interpolates the received values from each server, which result in $Iresult$, Iop_1, and Iop_2, as follows:

$$Iresult = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \times 44 + \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \times 145 + \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} \times 304 + \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} \times 521 = 1$$

$$Iop_1 = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \times 162 + \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \times 937 + \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} \times 2812 + \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} \times 6273 = 1$$

$$Iop_2 = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} \times -9 + \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} \times -107 + \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} \times -363 + \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} \times -849 = 3$$

Note that the user obtains: $Iresult = Iop_1$ and $Iop_2 = n$, where n is the number of tuples in the relation, and it is known to the user. Thus, it is proved that the servers followed the count query algorithm.

B. OTHER OPERATIONS RELATED TO A MAXIMUM QUERY

In this section, we consider two more cases of a maximum query, where the maximum value can occur in multiple tuple (§B.1) and find the maximum value (or retrieve the tuple having the maximum value) over a dataset outsourced by multiple DB owner (§B.2).

Value	SMR (o)	Function f_1	$1-o$	Function f_2
A	$(2,3) \otimes (2,2) = 10$	$2 \times 10 = 20$	$1-10=-9$	$4\times-9=-36$
B	$(4,3) \otimes (2,2) = 14$	$3 \times 14 = 42$	$1-14=-13$	$6 \times -13 = -78$
A	$(6,4) \otimes (2,2) = 20$	$5 \times 20 = 100$	$1-20=-19$	$3 \times -19 = -57$
	$result_1 = 44$	$\langle op_1 \rangle_1 = 162$		$\langle op_2 \rangle_1 = -9$

Table 15: Server 1 execution.

Value	SMR (o)	Function f_1	$1-o$	Function f_2
A	$(3,6) \otimes (4,3) = 30$	$3 \times 30 = 90$	$1-30=-29$	$7 \times -29 = -203$
B	$(8,5) \otimes (4,3) = 47$	$5 \times 47 = 235$	$1-47=-46$	$11 \times -46 = -506$
A	$(11,8) \otimes (4,3) = 68$	$9 \times 68 = 612$	$1-68=-67$	$5 \times -67 = -335$
	$result_2 = 145$	$\langle op_1 \rangle_2 = 937$		$\langle op_2 \rangle_2 = -107$

Table 16: Server 2 execution.

Value	SMR (o)	Function f_1	$1-o$	Function f_2
A	$(4,9) \otimes (6,4) = 60$	$4 \times 60 = 240$	$1-60=-59$	$10 \times -59 = -590$
B	$(12,7) \otimes (6,4) = 100$	$7\times100 = 600$	$1-100=-99$	$16 \times -99 = -1584$
A	$(16,12) \otimes (6,4) = 144$	$13 \times 44 = 1872$	$1-144=-143$	$7 \times -143 = -1001$
	$result_3 = 304$	$\langle op_1 \rangle_3 = 2812$		$\langle op_2 \rangle_3 = -3175$

Table 17: Server 3 execution.

Value	SMR (o)	Function f_1	$1-o$	Function f_2
A	$(5,12) \otimes (8,5) = 100$	$5\times100 = 500$	$1-100=-99$	$13 \times -99 = -1287$
B	$(16,9) \otimes (8,5) = 173$	$9 \times 173 = 1557$	$1-173=-172$	$21 \times -172 = -3612$
A	$(21,16) \otimes (8,5) = 248$	$17 \times 248 = 4216$	$1-248=-247$	$9 \times -247 = -2223$
	$result_4 = 521$	$\langle op_1 \rangle_4 = 6273$		$\langle op_2 \rangle_4 = -7122$

Table 18: Server 4 execution.

B.1 Multiple Occurrences of the Maximum Value

In practical applications, more than one tuple may have the maximum value in an attribute, *e.g.*, two employees (E103 and E015) earn the maximum salary in design department; see Figure 4. However, the above-mentioned methods (for `QMax1` or `QMax2`) cannot fetch all those tuples from the relation $S(R^1)$ in one round. The reason is that since the cloud i uses OP-SS values of the attribute A_c of the relation $S(R^2)_i$ for finding the maximum value, where more than one occurrences of a value have different representations, the cloud i cannot find all the tuples of $S(R^2)$ having the identical maximum value, by looking OP-SS values.

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

Data outsourcing. The DB owner outsources the relation $S(R^1)$ as mentioned in §3.1. However, the DB owner outsources the relation $S(R^2)$ with four columns: `CTID`, `SSTID`, `OP-SS-`A_c, and `SS-`A_c. The first three columns are created in the same way as mentioned in §3.1. The i^{th} value of `SS-`A_c attribute has the same value as the i^{th} value of `OP-SS-`A_c attribute. However, this value is secret-shared using the unary representation, as the column A_c of the relation $S(R^1)$ has. However, the DB owner uses different polynomial over the i^{th} value of the attribute A_c of $S(R^1)$ and the attribute `SS-`A_c of $S(R^2)$, so that the adversary cannot related two relations.

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

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

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

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

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

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

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

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

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

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

Note. Answering conditional maximum query. We are not providing details for fetching all the tuples having the maximum in the case of conditional maximum queries. For answering a conditional maximum query, the user include the above-mentioned two steps to the method given in §7.2. Thus, fetching all tuples having the maximum value in the attribute A_c requires three communication rounds, and each cloud scans the entire relation $S(R^1)$ three times. In particular, in the first round, the cloud i provides `Index` values to the user. In the second round, the cloud i finds the tuple having the maximum value in the attribute A_c from the requested tuple ids, implements the above-mentioned method given in round 1, and provides a list of n numbers. In the last round, the user fetches all the desired tuples.

B.2 Finding Maximum over Datasets Outsourced by Multiple DB Owners

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

Data outsourcing. Consider that an i^{th} DB owner wishes to outsource a number, say v. The i^{th} DB owner creates shares of v using a secret-sharing mechanism (either HSS or AA) that allows string-matching operations at the cloud and sends to the c non-communicating clouds, as described in §3.1. However, note that, here, we do not outsource numbers using the unary representation, which was used for other queries in previous sections. In this case, the DB owner first creates a binary representation of the number and then creates the shares. Binary representation allows us to execute 2's complement-based signbit computation, as follows:

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

Thus, for each number at the i^{th} position, say V_i, the cloud j computes the signbit with all the other numbers using 2's complement-based subtraction, *i.e.*, $signbit(V_i - V_x)$, $x \neq i$, and $1 \leq x, i \leq n$. Recall that the signbit results in 1 of secret-share form, if $V_i < V_x$; otherwise, 0. Then, the cloud j adds all $n-1$ signbit values computed for the i^{th} share of the list A_c. Therefore, after comparing each pair of inputs and adding corresponding $n-1$ signbit values, the cloud j has a vector, say vec, of n shares. The user asks the count query (§5) to find the occurrences of 0 in vec (it will be clear soon why the user is asking for counting 0) and the sum of the values of A_c for which the count query resulted in 1 of secret-shared form.

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

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

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

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

C. MINIMUM, TOP-K, AND GROUP-BY

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

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

```
QMin1. select * from Employee where Salary
    in (select min(Salary) from Employee)
QMin2. select * from Employee as E1 where
  E1.Dept = 'Testing' and Salary in (select
    min(salary) from Employee as E2 where
            E2.Dept = 'Testing')
```

Here, in short, we explain how to execute these queries on the relations $S(R^1)$ and $S(R^2)$, since these queries are similar to maximum queries §7. To execute an unconditional minimum query, the user follows the same strategy for solving `QMax1` (§7.1); however, the user asks for the minimum value from the relation $S(R^1)$. First, each cloud i finds a tuple, say $\langle S(t_k), S(value)\rangle_i$, where $S(t_k)_i$ is the k^{th} secret-shared tuple id (in the attribute `SSTID`) and $S(value)_i$ is the secret-shared minimum value of the A_c attribute in the k^{th} tuple. Finally, the cloud i compares the tuple id $\langle S(t_k)\rangle_i$ with each k^{th} value of the attribute `TID` of $S(R^1)_i$ and multiplies the resultant by the first m attribute values of the tuple k. Finally, the cloud i adds all the values of each m attribute.

To execute a conditional minimum query, the user operates in two rounds, like a conditional maximum query; see §7.2. In the round 1, the user obliviously knows the tuple ids of the relation $S(R^1)$ satisfying query predicate. In round 2, the user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the clouds. Each cloud i finds the minimum value of the attribute A_c in the requested tuple ids by looking into the attribute `CTID` of the relation $S(R^2)_i$ and results in a tuple, say $\langle S(t_k), S(value)\rangle_i$, where $S(t_k)_i$ shows the secret-shared tuple id (from `SSTID` attribute) and $S(value)_i$ shows the secret-shared minimum value. Finally, the cloud i performs a join operation between all the tuples of $S(R^1)_i$ and $\langle S(t_k), S(value)\rangle_i$, as performed when answering unconditional maximum (`QMax1`) queries; see §7.1.

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

Top-k. We again consider unconditional and conditional queries in the case of a top-k query. In both the cases, the user follows a similar approach, like maximum queries; see §7; however, the user asks for top-k values instead of the maximum value.

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

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

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

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

Conditional top-k query. Answering conditional top-k queries require when all the values of the attribute A_c are unique requires two communication rounds between the user and the clouds, like a conditional maximum query, see §7.2, as follows:

Round 1. The user obliviously knows the tuple ids of the relation $S(R^1)$ satisfying the query predicate.

Round 2. The user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the clouds. Each cloud i finds the top-k values of the attribute A_c in the requested tuple ids by looking into the attribute CTID of the relation $S(R^2)_i$ and results in a set of k tuples. Now, the cloud i performs a join operation between all the tuples of $S(R^1)_i$ and each of the k tuples of the relation $S(R^2)$, as performed above in answering an unconditional top-k query.

Note. A reverse-top-k query can also be executed in a same manner like top-k queries; however, the user asks for the minimum-k values.

Group-by. A group-by query finding the counting/sum of the values in any attribute can be executed like a count/sum query; see §6. However, the user needs to know the name of the groups, and then, the user sends the group names to the servers and attribute on which the user wants to execute a group-by query. Note that it may reveal the number of groups in an attributes; and hence, the user may ask the query for some fake groups.

Consider the following group-by query: `select count(*) from` R `group by` A_i. For answering this group-by query, the server j executes the following computation on each tuple of the relation R for each group:

$$Output_l = \sum_{k=1}^{k=n}(A_i[S(a_k)]_j \otimes S(v_l)_j)$$

Where $1 \leq l \leq g$, v_l is the name of each group, $\otimes$ shows a string-matching operation, and $Output_l$ is the answer to the group-by query.

Consider the following group-by query involving sum operation: `select sum(`A_ℓ`) from` R `group by` A_i. For answering this group-by query, the server j executes the following computation on each tuple of the relation R for each group:

$$Sum_l = \sum_{k=1}^{k=n} A_\ell[S(a_k)]_j \times (A_i[S(a_k)]_j \otimes S(v_l)_j)$$

Where $1 \leq l \leq g$, v_l is the name of each group, $\otimes$ shows a string-matching operation, and Sum_l is the answer to the group-by query.

D. METHODS FOR FINDING TUPLE IDS

A trivial solution for knowing the tuple ids satisfying a query predicate is given in §7.2 that transmits n numbers from each cloud to the user.

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

The first method. The cloud creates $\mathcal{T}$ columns,[14] one for each tuple id that satisfies the query predicate, say v. Note that actually we do not need to create any column during implementation, we need to have $\mathcal{T}$ variables. For the purpose of explanation, we show $\mathcal{T}$ columns. Each column has allocated one of the values from 1 to $\mathcal{T}$ of secret-share form (provided by the user). After an oblivious computation over each tuple, if there are $\mathcal{T}$ occurrences of v, then each of the $\mathcal{T}$ columns will have one of the exact tuple id where v occurs. The cloud executes the following operation:

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

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

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

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

Example. Table 19 shows an implementation of tuple id finding method in cleartext to know the tuple ids that have `Dept = Testing`; see Figure 5a for Employee relation. Note that for each row, we perform string-matching operations whose results are stored in the variable o, and all the occurrences of the query predicate are stored in the variable a. The user asks the cloud to create two columns ($\mathcal{T} = 2$) for keeping tuple ids.

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

The second method. Now, we propose a method that takes at most three communication rounds, and in each round, each cloud sends

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

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

Table 19: Knowing tuple ids of employees working in testing department.

$\sqrt{n}$ numbers to the user.

After performing the string-matching operation on the shares, we get a vector of n values that have either 1 or 0 of secret-share form. Thus, sending these n numbers are enough to reveal all the positions of a predicate in the database. For example, in Table 4, if the user wants to know the row-ids of all Johns, the clouds only need to return the vector $\langle 1, 1, 0, 0, 0, 0 \rangle$ of secret-share form. The user can fetch the first and the second tuple obliviously be following the method given in §7.2.[15] However, the scale of such a vector is equal to the number of rows in a database.

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

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

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

Algorithm 2: $Eval(U, V)$

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

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

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

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

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

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

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

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

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

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

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

writes the matrix as follows:

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

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

$$
\mathbf{M} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & X & 0 & X & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}. \quad (3)
$$

Now, the user checks the number of uncertain positions, which are reduced to 6. Since the user knows there are seven 1s in the matrix, the user replaces all X with 1 and recover the database.
Round 3: In this case, no third round of communication is needed. But sometimes if the user cannot recover all the occurrences' positions, the user will ask the clouds to send the data in all possible positions, which reveal the answer immediately.

E. SECURITY PROOF OUTLINE

Now, we provide the security proof outline for OBSCURE. In our context, we, first, need to show that an adversary cannot distinguish any two queries of the same type based on the output size, *i.e.*, the query/user privacy will be maintained. Once we can prove the query privacy, we will show how the server privacy (*i.e.*, not revealing more information to the user) is achieved.

Theorem 1 *If the adversarial cloud can distinguish two input queries, then either the random polynomials used for creating shares of a query is not correct or* OBSCURE *does not provide query privacy.*

In order to show that the adversary can never know the exact query value, we consider two instances of the datasets, as follows: D_1 and D_2, where D_1 differs from D_2 only at one value each, say v_1 and v_2, *i.e.*, v_1 is in D_1 but D_2 and v_2 is in D_2 but D_1. Here, we show that if the adversary can distinguish the single different value in D_1 and D_2, she can break OBSCURE. In this setting, the cloud executes the input queries on D_1 and D_2.

By our assumption of ciphertext indistinguishability (mentioned in §3.3), the adversary cannot distinguish that D_1 and D_2 are identical or different. Note that if the DB owner uses only one polynomial (*i.e.*, a weak cryptographic plan), then the adversary can find which value is the only single values of D_1 that is different from values of D_2. Moreover, it reveals frequency-count of values.

Now assume the queries for the value v_1 and v_2 that will be mapped to secret-shared queries, $q_{v1}(D_1)$ and $q_{v1}(D_2)$, respectively. Further, assume that $q_{v1}(D_1)$ and $q_{v2}(D_2)$ are identical. Hence, the adversary will consider both of them as an identical query, while they are for different queries. Hence, the adversary cannot distinguish two queries. Now, assume that $q_{v1}(D_1)$ and $q_{v2}(D_2)$ are different, and here the adversary objective is to deduce which tuple of relations satisfy the query or not. If the adversary cannot know which tuple is satisfying the query or not, the adversary can distinguish two queries, as well as, the two dataset. This violates our assumption of ciphertext indistinguishability of the dataset. Thus, the adversary cannot distinguish two datasets or two queries.

Now, we provide an intuition that how does the server privacy is maintained. Recall that we assumed a trusted user. In response to a query, the user obtains a some numbers. Since the servers cannot distinguish between two queries and they follow the algorithm on the entire dataset,

F. BUCKETIZATION-BASED RANGE QUERIES

As we mentioned that we convert a range query into several point queries that cover the entire range. However, as per Exp 8 (Figure 5), as the range increases, the computation time also increases. In order to reduce the computation time, we propose a new method that creates bins over the domain of attribute values and organizes these bins into a k-way tree, where k is the number of child nodes of a node or the number of values in each node at the lowest level. The bucketization-based range queries works as follows:
DB owner. Assume that the domain of values in an attribute has $1, 2, \ldots, n$ numbers. The DB owner first creates a k-way tree, by creating n/k nodes at the 0^{th}-level by placing $1, 2, \ldots, k$ numbers in the first node, $k+1, k+2, \ldots, 2k$ numbers in the second node, and so on. The first level node has $n/2k$ nodes, where the first node of the first level becomes parent of the first k nodes of 0^{th} node. The second node of the first level becomes a parent of $k+1, k+2, \ldots, 2k$ nodes of 0^{th}-level. In this way, the DB owner construct a k-way tree of height $\log_k(n/k)+1$. Now for each level, except the root node, the DB owner adds one attribute in the relation R. An i^{th} value of the attribute corresponding to any level, say j, except the root and 0^{th}-level keeps the j^{th} level's node id that covers the i^{th} value of the level 0.

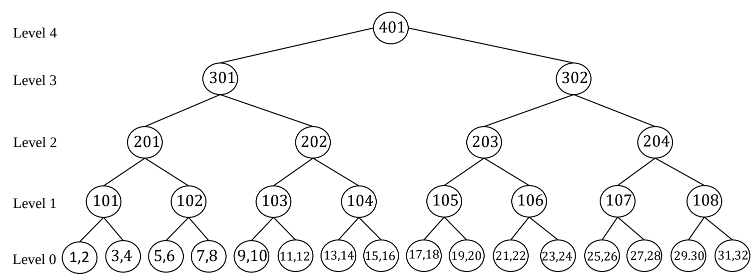


Figure 6: 2-way tree for 32 values.

Assume that an attribute A of a relation R has 32 numbers $(1, 2, \ldots, 32)$.[16] Here, we show how does the DB owner create a 2-way tree and three additional columns. Figure 6 shows a 2-way tree for 32 numbers. In a 2-way tree, the 0^{th} level has $n/k = 16$ nodes,

[16]For simplicity of presentation, we assume that the attribute has 32 continuous numbers. Having any 32 numbers will not affect the algorithm. In the case of any 32 number, we will create k-way tree for the minimum and maximum value in the domain, so that the resulting tree will have many empty nodes.

each with two numbers. The tree height is $\log_k(n/k) + 1 = 5$. Here, the DB owner adds three columns, say A_1, A_2, and A_3, in the relation for levels 1, 2, and 3 of the tree; see Table 20. Note that, for example, 9^{th} value of the attributes A_1, A_2, and A_3 contains node-ids of the respective levels that cover 9^{th} value of the level 0. Thus, the attribute A_1 contains 103, since `Node 103` covers the value 9, the attribute A_2 contains 201, since `Node 201` covers the value 9, and the attribute A_3 contains 301, since `Node 301` covers the value 9.

A	A_1	A_2	A_3
1	101	201	301
2	101	201	301
3	101	201	301
4	101	201	301
5	102	201	301
6	102	201	301
7	102	201	301
8	102	201	301
9	103	202	301
10	103	202	301
11	103	202	301
12	103	202	301
13	104	202	301
14	104	202	301
15	104	202	301
16	104	202	301
17	105	203	302
18	105	203	302
19	105	203	302
20	105	203	302
21	106	203	302
22	106	203	302
23	106	203	302
24	106	203	302
25	107	204	302
26	107	204	302
27	107	204	302
28	107	204	302
29	108	204	302
30	108	204	302
31	108	204	302
32	108	204	302

Table 20: A relation R having three new attributes, A_1, A_2, and A_3, based on bucketization of range values.

Creating secret-shares of the relation. The DB owner c secret-shares of each attribute value $A_i[a_j]$ of the relation R using a secret-sharing mechanism that allows string-matching operations at the server (as specified in §2).

User. We assume that the user is aware of the k value used in the k-way tree creation. For a given range, the user first finds the minimal set of nodes that cover the range, and then, creates secret-shares of those node values. We follow a least-match method for searching node value. Assume a query is for counting the number of tuples having values between `1` and `13`. The best-match method will find only `Node 301` that satisfies this query. However, it will cover some other values too, resulting in a wrong answer to the query. Thus, using a minimal set of nodes that cover the range, the user breaks the range into sub-ranges such as `Node 201`, `Node 103`, and value `13`. Note that by breaking the range from 1-13 into point queries requires to search for 13 different values. However, here using 2-way tree, the server will search only for three values.

Finally, the user creates secret-shares of these three values (`Node 201`, `Node 103`, and value 13) and sends them to the servers with the information of desired attribute on which the server should search for a value.

Server. The server execute the count query as mentioned in §5. Particularly, in this example, each server searches for `Node 201` in the attribute A_2, for `Node 103` in the attribute A_1, and for the value `13` in the attribute A. Finally, the server adds the outputs of all three individual searches, which produce the final answer to the count query.

Note. By following the same idea of breaking a range into sub-range, one can execute conjunctive and disjunctive count/sum queries.

Experiment. We pre-computed the range information with $k = 10$ for OrderKey values (with domain of 1-150000) of LineItem table of TPC-H. We outsourced the secret-shared version of this range information along with the original data. We executed 1D-count query on OrderKey values. Figure 7 shows that with pre-computed range information in the form of buckets significantly decreases the computation time amount of time spent in range based filtering over secret-shared data. Note that the naive range implementation scales linearly with the length of range, where as the bucketized pre-computed range information along with each tuple, takes almost constant time even after increasing the range length. Furthermore, it sometimes drops as fewer buckets are able to cover the entire range. For example, a range of 1-99 requires 19 searches (9 for 9 buckets covering values from 1 to 90 and extra 9 searches for values 91 to 99), whereas for a range from 1 to 100 we only need 1 bucket to represent it, therefore the time required to execute a range query for 1 to 100 decreases. Figure 8 shows how the number of multiplications per row vary with the increase in length of range. We can see that the naive implementation requires many multiplications as compared to the bucketization-based range algorithm. However, since we store range information along with each tuple, the size of the database increases, requiring more time to scan the table. Figure 7 shows that for a small ranges (length < 10), the naive algorithm performs better as the number of multiplication require by both the algorithms are equal, but the scanning time for the naive algorithm is smaller than the bucketization based algorithm.

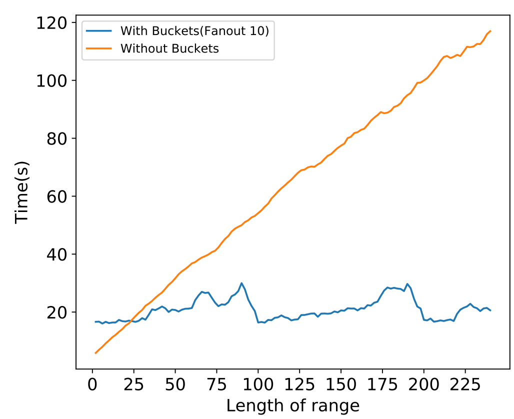


Figure 7: Impact of executing range queries using bucketization.

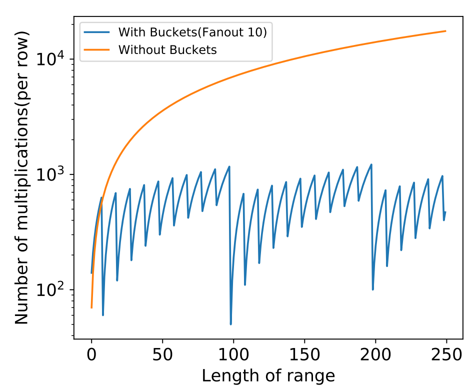


Figure 8: Number of multiplication operations used in string matching operation in range queries.