OBSCURE: Information-Theoretic Oblivious and Verifiable Aggregation Queries

Peeyush Gupta\textsuperscript{1}, Yin Li\textsuperscript{2}, Sharad Mehrotra\textsuperscript{1}, Nisha Panwar\textsuperscript{1}, Shantanu Sharma\textsuperscript{1}, and Sumaya Almanee\textsuperscript{1}
\textsuperscript{1}University of California, Irvine, USA. \textsuperscript{2}Xinyang Normal University, China.

ABSTRACT

Despite extensive research on cryptography, secure and efficient query processing over outsourced data remains an open challenge. This paper develops communication-efficient and information-theoretic secure algorithms for privacy-preserving aggregation queries using multi-party computation (MPC). More specifically, query processing techniques over secret-shared data outsourced by single or multiple database owners are developed. These algorithms allow a user to execute the 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 provide (non-mandatory) privacy-preserving result verification algorithms that detect some 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) \cite{33} allows authenticated users to execute their queries on an untrusted public cloud. Over the last two decades, several cryptographic techniques (e.g., SSS, MPC, and threshold cryptography, as mentioned in \cite{11}) have been proposed to achieve secure and privacy-preserving computations in the DaS model. These techniques can be broadly classified based on the cryptographic security into two categories: Computationally secure techniques that assume the adversary lacks adequate computational capabilities to break the underlying cryptographic mechanism in polynomial time. Non-deterministic encryption \cite{32}, homomorphic encryption \cite{31}, order-preserving encryption (OPE) \cite{8}, and searchable-encryption \cite{41} are examples of such techniques. Homomorphic encryption mixed with oblivious-RAM (ORAM) offers the most computationally secure mechanisms. Information-theoretic secure techniques that are unconditionally secure and independent of adversary’s computational capabilities. Shamir’s secret-sharing (SSS) \cite{40} is a well-known information-theoretic 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 mentioned in \cite{11}.

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

While both computationally and information-theoretic 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 \cite{33}) resulting in systems such as CryptDB \cite{39}, Monomi \cite{43}, MariaDB \cite{1}, CorrectDB \cite{10}. Some exceptions to the above include \cite{27, 28, 45, 26} that have focused on secret-sharing.

Recently, both academia and industries have begun to explore information-theoretic secure techniques using MPC that efficiently supports OLAP tasks involving aggregation queries, while achieving higher security than computationally secure techniques\cite{11}. For instance, commercial systems, such as Jana \cite{9} by Galois, Pulstar \cite{3} by Stealth Software, Sharemind \cite{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 \cite{29, 22, 38} and popular media \cite{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 \cite{27, 45, 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 generating shares at the database (DB) owner, no support for third-party query execution on the secret-shared outsourced database, etc. We discuss the limitations of existing secret-sharing-based data management techniques in details in \cite{22}.

Our contributions in this paper are twofold:

1. SSS-based algorithms (named as 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, while reveals nothing about

\textsuperscript{1}Some of the computationally secure mechanisms are vulnerable to computationally sufficiently powerful adversaries. For instance, Google, with sufficient computational capabilities, broke SHA-1 \cite{2}.
data/query/results to an adversary. Our algorithm supports aggregation queries on a dataset outsourced by a single DB owner (e.g., patient database outsourced by a single hospital) or multiple DB owners (e.g., data from smart meters).

2. An oblivious result verification algorithm for aggregation queries, and ensuring that an adversary does not learn anything about the verification. OBSCURE’s verification step is not mandatory and depends on the querier’s choice.

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 algorithms with some existing works.

2.1 Building Blocks

Our proposed algorithms are based on Shamir’s secret-sharing (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 [40], the DB owner divides a secret value, say S, into c different fragments, called shares, and sends each share to a set of c non-communicating participants/servers. These servers cannot know the secret S until they collect c’ < c shares. In particular, the DB owner randomly selects a polynomial of degree c’ with c’ random coefficients, i.e., \( f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{c’}x^{c’} \), where \( f(x) \in F_p[x] \), \( p \) is a prime number, \( F_p \) is a finite field of order \( p \), \( a_0, a_1, \ldots, a_{c’} \in \mathbb{N}(1 \leq i \leq c’) \). 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 \( c’ \) to be identical.

SSS allows an addition of shares, i.e., if \( s(a_1) \) and \( s(b_1) \) 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., \( a + b = s(a_1) + s(b_1) \), 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, 35, 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 alphabets has only three alphabets, namely A, B, and C. Thus, A can be represented as \((1, 0, 0)\). Similarly, B and C can be represented as \((0, 1, 0)\) and \((0, 0, 1)\), 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: \((0, 1, 0)\). If the DB owner outsources the vector \((0, 1, 0)\) to the servers, it will reveal the alphabet. Thus, the DB owner uses any three polynomials of an identical degree, as shown in Table 1 to create three shares.

```
<table>
<thead>
<tr>
<th>Vector values</th>
<th>Polynomials</th>
<th>First shares</th>
<th>Second shares</th>
<th>Third shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 + ox</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>1 + ox</td>
<td>15</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0 + 2x</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Table 1: Secret-shares of a vector \((0, 1, 0)\), created by the DB owner.

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

```
<table>
<thead>
<tr>
<th>Vector values</th>
<th>Polynomials</th>
<th>First shares</th>
<th>Second shares</th>
<th>Third shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 + x</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1 + 2x</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0 + 4x</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>
```

Table 2: Secret-shares of a vector \((0, 1, 0)\), 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.

```
<table>
<thead>
<tr>
<th>Computation on</th>
<th>Server 1</th>
<th>Server 2</th>
<th>Server 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 \times 2 = 10</td>
<td>10 \times 2 = 20</td>
<td>15 \times 2 = 30</td>
<td></td>
</tr>
<tr>
<td>10 \times 2 = 20</td>
<td>10 \times 2 = 20</td>
<td>20 \times 2 = 40</td>
<td></td>
</tr>
<tr>
<td>2 \times 8 = 16</td>
<td>4 \times 8 = 32</td>
<td>6 \times 12 = 72</td>
<td></td>
</tr>
</tbody>
</table>
```

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

**User-side.** After receiving the outputs \((y_1 = 43, y_2 = 147, y_3 = 313)\) from the three servers, the user executes Lagrange interpolation [20] to construct the secret answer, as follows:

\[
\begin{align*}
(y_1 - x_2)(y_2 - x_1) & \times y_3 + (y_2 - x_3)(y_3 - x_2) & \times y_1 + (y_3 - x_1)(y_1 - x_3) & \times y_2 \\
& = (43 - 2)(147 - 3) & \times 313 + (147 - 3)(313 - 2) & \times 43 + (313 - 2)(43 - 3) & \times 147 + (2 - 3)(147 - 2) & \times 313 = 1
\end{align*}
\]

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 Pri[21] also uses a unary representation; however, we use significantly less number of bits compared to Pri’s unary representation. One can use Pri’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 2 on page 5. For explanation purpose, we represent Table 2 in cleartext. In Table 2, 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 sigbit to find if two numbers are equal or not, as follows: $A \geq B$ if $\text{sigbit}(A - B) = 0$, and $A < B$ if $\text{sigbit}(A - B) = 1$. Let $A = [a_0, a_1, \ldots, a_n]$ be a $n$ bit number and $B = [b_0, b_1, \ldots, b_1]$ 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 $a_0 + b_0 + 1$ and its carry bit. After finishing this on all the $n$ bits, the most significant bit (MSB) keeps the sigbit.

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 not secure due to using OP-SS mechanism to answer maximum/minimum queries. Another paper by Emekçi et al. [28] also focused on aggregation queries based on OP-SS. However, [28] requires the database (DB) owner to retain each polynomial, which was used to create shares of the database. Thus, the DB owner stores $n \times m$ polynomials, where $n$ and $m$ are the numbers of tuples and attributes in a relation. Also, [28] is not secure due to revealing access-patterns (i.e., the identity of tuples that satisfy a query) and using OP-SS. Like [28], [45] proposed a similar approach and also suffers from similar disadvantages. [42] proposed SSS-based sum and average queries; however, they also require the DB owner to retain tuple ids of qualifying tuples. [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 works 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, OBSOURE eliminates all such limitations. Moreover, as mentioned previously, OP-SS is not secure, especially, prone to background knowledge attacks. OBSOURE uses OP-SS for answering maximum and minimum in such a way that prevents background knowledge attacks on OP-SS by partitioning the data (which will be clear soon in [41], unlike [27, 24, 28]. In addition, it is important to mention that for string-matching operations on shares, we do not develop any new algorithm. We use the string-matching algorithm given in [24] that performs search operations on shares. Furthermore, as we will see in experimental section ([8], OBSOURE scales to datasets with 6M tuples on TPC-H queries.

Comparison with MPC-techniques. OBSOURE also overcomes several limitations/disadvantages of existing MPC-based solutions. Recent work by Corrigian-Gibbs and Boneh’s, 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 OBSOURE is designed to only handle malicious clouds, 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, OBSOURE uses minimal communication rounds between the user and each server, (when having enough shares). Specifically, count, sum, average, and their verification algorithms require at most two rounds between each server and the user. However, maximum/minimum finding algorithms require at most four communication rounds. In addition, our scheme achieves the minimum communication cost for aggregate queries, especially for count, sum, and average queries, by aggregating data locally at each server.

Comparison with MPC/SSS-based verification approaches. [35] and [42] provided verification approaches for secret-shared data. [35] considered verification process for MPC using a trusted-third-verifier. [42] provided operation verification (i.e., whether all the desired tuples are scanned or not) for only sum queries, unlike OBSOURE that verifies results for all queries. Also, [42] overburdens the DB owner by keeping metadata for each tuple, which serves as a mechanism for sum verification. OBSOURE verification methods neither involve the DB owner to verify the results nor require a trusted-third-verifier.

3. PRELIMINARY

This section provides a description of entities, an adversarial model, and security properties for obliquely 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 $i^{\text{th}}$ share to the $i^{\text{th}}$ 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 [21].

3. An (authenticated, authorized, and trusted) user/querier, who executes queries on the secret-shared data at the server. 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 \cite{20}) to obtain the secret-value.

### 3.2 Adversarial Model

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

The first type of adversarial behavior is identical to an honest-but-curious adversary, which is considered widely in many cryptographic algorithms and in the standard DaS query processing model, keyword searches, and aggregation queries \cite{18,33,44,46}. In literature, the second type of adversarial behavior is considered under a malicious adversary. Note that we do not assume any malicious user and malicious DB owners.

We follow the restriction of the standard SSS that the adversary cannot collude all (or possibly the majority of) the servers. Thus, the adversary cannot generate/insert/update shares on the majority of the servers. Also, the majority of communication channels between all the servers and the user are unknown to the adversary. The adversary can eavesdrop on the minority of the communication channels between the server and the DB owner/user to gain knowledge about data, queries, or results. The channel is assumed to be secure, and only authenticated users can request a query on the servers. The user is trusted and never behaves maliciously. The adversary is also aware of some public information, such as the actual number of tuples and number of attributes in a relation, which can be hidden by adding fake tuples and attributes.\footnote{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.}

### 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 definitions in \cite{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 \(q_1\) and \(q_2\), the server cannot distinguish \(q_1\) or \(q_2\) based on the executed computations for either \(q_1\) and \(q_2\).

**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 \(i.e.,\) following the approach given in \cite{24,71}, 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 \(i.e.,\) recall that user might be different compared to the data owner in our model). To show that we need to compare the real execution of the algorithm at the servers against the ideal execution of the algorithm at a trusted party having the same data and the same query predicate. An algorithm maintains the data privacy from the server if the real and ideal executions of the algorithm return an identical answer to the user.

**Definition: Privacy from the server.** For any given secret-shared relation \(S(R)\) at a server, any query predicate \(q_p\), 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 \(q_p\) 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 \cite{35}, as follows: (i) the verification method cannot be refuted by the majority of the malicious servers, and (ii) the verification method should not leak any additional information.

**Algorithms’ performance.** We analyze our oblivious aggregation algorithms on the following parameters, which are stated in Table\[7] (i) Communication rounds. The number of rounds that is required between the user and each server to obtain an answer to the query. (ii) Computational cost at the server. We measure the computational 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 OBSCUR3 Overview

Let us introduce OBSCUR3 at a high-level. OBSCUR3 allows single dimensional equality queries and multi-dimensional conjunctive/disjunctive queries. Note that the method of OBSCUR3 for handling these types of queries is different from SQL, since OBSCUR3 does not support query optimization and indexing due to secret-shared data. Further, OBSCUR3 handles range-based queries by converting the range into equality queries. Executing a query on OBSCUR3 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 \(e.g.,\) \cite{24,15,15} 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 \cite{3}(count query), \cite{6}(sum queries), \cite{7}(maximum/minimum), and \cite{5,6,14}(verification).

**Phase 3: Query processing by the servers.** The servers process an input query in an oblivious manner such that they do not learn
the query, as well as, the results satisfying the query. Finally, the 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 \( A_i \), on which a comparison operator (minimum, maximum, and top-k) needs to be supported.

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 \( A_i \) of the relation \( R^2 \) keeps the \( i^{th} \) value of an attribute of the relation \( R^1 \) on which the user wants to execute a comparison operator. Further, the tuples of the relations \( R^2 \) are randomly permuted. The reason of doing permutation is that the adversary cannot relate any tuple of both the secret-shared relations, which will be clear soon by the example below.

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

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>E101</td>
<td>John</td>
<td>1000</td>
<td>testing</td>
</tr>
<tr>
<td>E101</td>
<td>John</td>
<td>100000</td>
<td>Security</td>
</tr>
<tr>
<td>E102</td>
<td>John</td>
<td>2000</td>
<td>Design</td>
</tr>
<tr>
<td>E103</td>
<td>Eve</td>
<td>2000</td>
<td>Design</td>
</tr>
<tr>
<td>E104</td>
<td>Eve</td>
<td>1500</td>
<td>Design</td>
</tr>
<tr>
<td>E105</td>
<td>Mike</td>
<td>2000</td>
<td>Design</td>
</tr>
</tbody>
</table>

Table 4: A relation: Employee.

**Example.** Consider the Employee relation (see Table 4). The DB owner creates \( R^1 = \text{Employee1} \) relation [7] (see Table 5a) with TID and Index attributes. Further, the DB owner creates \( R^2 = \text{Employee2} \) relation (see Table 5b) having three attributes CTID, SSTID, and Salary.

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
<th>TID</th>
<th>Index</th>
<th>CTID</th>
<th>SSTID</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E101</td>
<td>John</td>
<td>1000</td>
<td>testing</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1500</td>
</tr>
<tr>
<td>E101</td>
<td>John</td>
<td>100000</td>
<td>Security</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5000</td>
</tr>
<tr>
<td>E102</td>
<td>John</td>
<td>2000</td>
<td>Design</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>E103</td>
<td>Eve</td>
<td>2000</td>
<td>Design</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2000</td>
</tr>
<tr>
<td>E104</td>
<td>Eve</td>
<td>1500</td>
<td>Design</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7000</td>
</tr>
<tr>
<td>E105</td>
<td>Mike</td>
<td>2000</td>
<td>Design</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 5: Two relations obtained from Employee relation.

1 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.

2 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.

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

Further, on the relation \( R^2 \), the DB owner creates \( c \) secret-shares of each value of SSTID using a secret-sharing mechanism that allows string-matching operations on the servers and each value of \( A_i \) using order-preserving secret-sharing [27][34][28]. The secret-shares of the relation \( R^2 \) are denoted by \( S(R^2)_i (1 \leq i \leq c) \). The attribute CTID is outsourced in cleartext with the shared relation \( S(R^2)_k \). It is important to mention that CTID attribute allows fast search due to cleartext representation than SSTID attribute, which allows search over shares.

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 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 \( A_1 = v_1 \) \& \& \( A_2 = v_2 \) \& \& \ldots \& \& \( A_m = v_m \).

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 \( A_1 = S(v_1)_j \) \& \& \( A_2 = S(v_2)_j \) \& \& \ldots \& \& \( A_m = S(v_m)_j \).
We explain the above conjunctive count query method. Each server \( j \) performs the following operations:

\[
Output = \sum_{k=1}^{m} \prod_{i=1}^{n} (A_{i}[S(a_{k})]_i \otimes S(v_{i,j}))
\]

\( \otimes \) 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 whose results will be 0 or 1 of secret-share form. Each server \( j \) compares the query predicate value \( S(v_{i,j}) \) 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 \( S(R^j) \) 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 6): select count(*) from Employee where Name = ‘John’ and Salary = ‘1000’. Table 6 shows the result of the private string-matching on the attribute Name, denoted by \( o_1 \), and on the attribute Salary, denoted by \( o_2 \). Finally, the last column shows the result of the query for each row and the final 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.

**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 \( A_1 = S(v_1) \lor A_2 = S(v_2) \lor \ldots \lor A_m = S(v_m) \)

The user transforms the query predicates to \( c \) secret-shares that results in the following query at the \( j^{th} \) server: select count(*) from \( S(R^j) \) where \( A_1 = S(v_{1,j}) \lor A_2 = S(v_{2,j}) \lor \ldots \lor A_m = S(v_{m,j}) \). The server \( j \) performs the following operation:

\[
Result^j_k = A_i[S(a_k)]_i \otimes S(v_{i,j}), 1 \leq i \leq m
\]

\[
Output = \sum_{k=1}^{m}((Result^j_1 \lor Result^j_2) \lor Result^j_3) \ldots \lor Result^j_m)
\]

Table 6: An execution of the conjunctive count query.

<table>
<thead>
<tr>
<th>Name</th>
<th>( o_1 )</th>
<th>Salary</th>
<th>( o_2 )</th>
<th>( o_1 \times o_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1</td>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>1</td>
<td>10000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adam</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Eve</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Alice</td>
<td>0</td>
<td>1500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mike</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7: Complexities of the algorithms.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Query conditions</th>
<th>Scan rounds at a server</th>
<th>Comm. rounds</th>
<th>Interpolated values at user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count [5]</td>
<td>SD</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DD</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sum [6]</td>
<td>SD</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DD</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Unconditional</td>
<td>max/min.</td>
<td>1 occurrence with tuple</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Condition</td>
<td>max/min/minimum</td>
<td>Finding maximum</td>
<td>( n + 1 )</td>
<td>6( \sqrt{n} ) + 1, or ( T + 1 )</td>
</tr>
<tr>
<td></td>
<td>(2DBMax [7,2])</td>
<td>Tuple fetching</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The following operations are given in full version [10].

| Maximum/Minimum | one/multiple occurrences | Counting | 2\( n + 1 \) | 1 | 2 |
|                | (2DBMax [7,2])           | Counting + tuple fetching | 2\( n + 3 \) | 3 | 2\( T + m \) |

| Maximum/Minimum | Many occurrences | Finding | 2 | 2 | 4 |
|                | (2DBMax [7,2]) | Tuple fetching | 4 | 4 | 2\( T + m \) |

<table>
<thead>
<tr>
<th>Group by</th>
<th>Top k or reverse</th>
<th>Unique occurrence</th>
<th>1 or k</th>
<th>2 or 1</th>
<th>k ( m )</th>
</tr>
</thead>
</table>

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

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.
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:
\[
op_1 = f_1(x) = \sum_{i=1}^{n} (S(a_i) \ominus o_i)
\]
\[
op_2 = \ op_1 + f_2(y) = \ op_1 + \sum_{i=m}^{n} f_2(y_i) (1-o_i) - 1
\]
i.e., the server performs the functions \( f_1 \) and \( f_2 \) on some n values, (it will be clear soon what are these n values). In the above equations \( o_i \) is the output of the string-matching operation carried on the \( i \)th value of an attribute, say \( A_i \), 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 \( R^3 \). 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 3, it executes the private string-matching operation on the \( i \)th (\( 1 \leq i \leq n \)) value of the attribute \( A_i \) against the query predicate \( \phi \) 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 \( A_j \) (and \( A_k \)) 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 \( A_i \), as a result, say \( \langle \text{result}\rangle_k \), 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 \( \text{Result} \), \( \text{Iop}_1 \), and \( \text{Iop}_2 \). If the server followed the algorithm, the user will obtain: \( \text{Result} = \text{Iop}_1 \) and \( \text{Iop}_2 = n \), where \( n \) is the number of tuples in the relation, and it is known to the user.

**Example.** We explain the above method using the following query on the Employee relation (refer to Table 4):
\[
\text{select count}(*) \text{ from Employee where Name = 'John'}
\]
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 \( \text{Name} \) attribute against the query predicate, the result of string-matching becomes 1 that is multiplied by the first value of the attribute \( A_i \), and results in 1. The complement of the resultant is multiplied by the first value of the attribute \( A_i \), and results in 0. All the other tuples are processed in the same way. Note that for this query, \( \text{Result} = \op_1 = 2 \) and \( \op_2 = 6 \), if server performs each operation correctly.

<table>
<thead>
<tr>
<th>Name</th>
<th>String-matching results</th>
<th>( f_1 )</th>
<th>( f_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Adam</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Eve</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Alice</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Mike</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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 \( \text{Result} = \text{Iop}_1 \) to be true. However, the second condition \( \text{Iop}_2 = n \) will never be true, since discarding one tuple will result in \( \text{Iop}_2 = n - 1 \). In the second case, the servers will send the wrong \( \text{Result} \) by discarding an \( i \)th count query resultant, and they will also discard the \( i \)th value of the attribute \( A_x \) to lead to \( \text{Result} = \text{Iop}_1 \) at the user-side. Here, the user, however, finds the second condition \( \text{Iop}_2 = n \) 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:
\[
\text{sum}(A_1) \text{ from } R \text{ where } A_1 = v_1 \land A_2 = v_2 \land \ldots \land A_m = v_m
\]
In the secret-sharing setting, the user transforms the above query into the following query at the \( j \)th server: \( \text{sum}(A_1) \text{ from } S(R^j) \), where \( A_1 = S(v_1) \land A_2 = S(v_2) \land \ldots \land A_m = S(v_m) \). 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 value of the attribute \( A_i \). The \( j \)th server performs the following operation on each attribute on which the user wants to compute the sum, i.e., \( A_t \) and \( A_q \):
\[
\sum_{k=1}^{n} A_t[iS[(A_j)]_k] \times ([\sum_{i=1}^{m} S(A_1[j]_i)] \lor S((v_i)))
\]
**Correctness.** The correctness of conjunctive sum queries is similar to the argument of the correctness of conjunctive count queries.

**Disjunctive sum query.** Consider the following query:
\[
\text{sum}(A_1) \text{ from } R \text{ where } A_1 = v_1 \lor A_2 = v_2 \lor \ldots \lor A_m = v_m
\]
The user transforms the query predicates to \( e \) secret-shares that results in the following query at the \( j \)th server:
\[
\text{sum}(A_1) \text{ from } S(R^j)
\]
where \( A_1 = S(v_1) \lor A_2 = S(v_2) \lor \ldots \lor A_m = S(v_m) \). The server \( j \) executes the following computation:
\[
\text{Result}_j = A_t[iS[(A_j)]_j] \lor S((v_i)), 1 \leq i \leq m, 1 \leq k \leq n
\]
\[
\text{Output} = \sum_{k=1}^{n} A_t[iS[(A_j)]_k] \times (((\text{Result}_1) \lor \text{Result}_2) \lor \ldots \lor \text{Result}_e)_j
\]
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., \( A_t \)), and then, adds all values of the attribute \( A_t \).

**Correctness.** The correctness of a disjunctive sum query is similar to the correctness of a conjunctive 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 are checked against the count query resultant, or \( \text{Iresult} = \text{Iop}_1 \).
values will be included by the following sum query. select sum($A_t$) from $R$ where $A_y = v$.

Here, our objective is to verify that (i) all tuples of the databases are checked against the sum query predicates, $A_y = v$, and (ii) only all qualified values of the attribute $A_t$ 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:

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

$$op_2 = f_2(x) = \sum_{i=1}^{n} o_i(y_i + a_i + o_i)$$

i.e., the server performs the functions $f_1$ and $f_2$ on some $n$ values, (it will be clear soon what are these $n$ values). In the above equations, $o_i$ is the output of the string-matching operation carried on the $i^{th}$ value of the attribute $A_y$, and $a_i$ be the $i^{th}$ ($1 \leq i \leq n$) value of the attribute $A_t$. 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 $R^i$. 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 $A_t$. 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 $A_y$ against the query predicate $v$ and multiplies the resultant value by the $i^{th}$ value of the attribute $A_t$. The server $k$ adds all the resultant values of the attributes $A_t$.

**Verify 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 $A_t$, say $\langle \text{sum}_{i=k} \rangle$, (ii) the sum of the outputs of the string-matching operation carried on the attribute $A_y$, say $\langle \text{sum}_{i=k} \rangle$, against the query predicate, and (iii) the sum of outputs of the functions $f_1$ and $f_2$, say $\langle \text{sum}_{f_1, f_2} \rangle$.

**User-side.** The user interpolates the received three values from each server, which results in $Isum_{i}$. $Isum_{i}$ and $Isum_{f_1, f_2}$. The user checks the value of $Isum_{f_1, f_2} - 2 \times Isum_{i}$ and $Isum_{e}$, and if it finds equal, the server has correctly executed the sum query.

**Example.** We explain the above method using the following query on the Employee relation (refer to Table 4): select sum(*) from Employee where Dept = 'Testing'. Table 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 $o_1$.

<table>
<thead>
<tr>
<th>Dept</th>
<th>Salary</th>
<th>$o$ values</th>
<th>$A_x$ and $f_1$</th>
<th>$A_y$ and $f_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>12000</td>
<td>0</td>
<td>$10000+12000+0$</td>
<td>$10000+12000+0$</td>
</tr>
<tr>
<td>Testing</td>
<td>20000</td>
<td>0</td>
<td>$10000+20000+0$</td>
<td>$10000+20000+0$</td>
</tr>
<tr>
<td>Design</td>
<td>1500</td>
<td>0</td>
<td>$1500+1500+0$</td>
<td>$1500+1500+0$</td>
</tr>
<tr>
<td>Design</td>
<td>2000</td>
<td>0</td>
<td>$2000+2000+0$</td>
<td>$2000+2000+0$</td>
</tr>
</tbody>
</table>

Table 9: An execution of the sum query verification.

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 $o_1$ (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, $Isum_{f_1, f_2} = 2 \times Isum_{i} = Isum_{e}$, i.e., $6004 \times 2 = 2 \times 6000$.

**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 $A_x$, $A_y$, and $A_q$.

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 finds $Isum_{f_1, f_2} - 2 \times Isum_{i} \neq Isum_{e}$, since an adversary cannot provide a wrong value of $Isum_{i}$, which is detected by count query verification. In order to hold the equation $Isum_{f_1, f_2} - 2 \times Isum_{i} = Isum_{e}$, the adversary needs to generate shares such that $Isum_{f_1, f_2} - Isum_{i} = 2 \times Isum_{i}$, but an adversary cannot generate any share, as 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 §4.2.

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 *, MAX(salary) from Employee

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

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 databases. 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 $A_y$ be an attribute of the relation $S(R^i)$ on which the user wishes to execute maximum queries. Our idea is based on a combination of OP-SS [27][34].

Note that we showed only a single dimensional condition in QMax2 query. Our proposed algorithms (without any modification) can find maximum/minimum while satisfying conjunctive and disjunctive conditions.
and SSS 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 [5.1], the attribute $A_i$ 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 $S(R^2)_i$. 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 1), 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 $(S(t_k), S(value))$, having the maximum value in the attribute $A_c$, where $S(t_k)$, is the $k^{th}$ secret-shared tuple id (in the attribute SSTID) and $S(value)$, is the secret-shared value of the attribute $A_c$ in the $k^{th}$ tuple.

2. **On the relation $S(R^1)$._** Now, the server $i$ performs the join of the tuple $(S(t_k), S(value))$, with all the tuples of the relation $S(R^1)_i$, by comparing the tuple ids (TID attribute values) of the relation $S(R^1)_i$, with $(S(t_k))$, as follows: 

$$
\sum_{k=1}^{m} A_p[S(a_k)] \times (TID[S(a_k)] \cup S(t_k))
$$

Where $p (1 \leq p \leq m)$ is the number of attributes in the relation $R$ and TID is the tuple id attribute of $S(R^1)$. To say, the server $i$ compares the tuple id $(S(t_k))$, 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 $(S(t_k))$, 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 $S(R^2)_i$, 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 $S(R^2)_i$. 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 $S(R^1)_i$ 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 $S(R^1)_i$ has the maximum value in the attribute $A_c$, due to executing an identical operation on each tuple of $S(R^2)$ 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 tuple ids of the relation $S(R^1)_i$ satisfying the where clause of the query (the method for obliviously finding the tuple ids is given below).

**Round 2.** The user interpolates the received tuple ids and sends the desired tuple ids in cleartext to the servers. Each server $i$ finds the maximum value of the attribute $A_c$ in the requested tuple ids by looking into the attribute CTID of the relation $S(R^1)_i$, and results in a tuple, say $(S(t_k), S(value))$, where $S(t_k)$ shows the secret-shared tuple id (from SSTID attribute) and $S(value)$ shows the secret-shared maximum value. Now, the server $i$ performs a join operation between all the tuples of $S(R^1)_i$, and $(S(t_k), S(value))$, as performed when answering unconditional maximum (QMax1) queries; see [3.1]. 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 tuple ids of $S(R^1)_i$ relation satisfying the where clause of a query in the case of conditional maximum queries.

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

**Information leakage discussion.** In round 1, due to obliviously retrieving tuple ids of $S(R^1)_i$, the adversary cannot know which tuples satisfy the query predicate. In round 2, the user sends only the desired tuple ids in cleartext to fasten the lookup of the maximum salary. Note that by sending tuple ids, the adversary learns the number of tuples satisfying the query predicate; however, the adversary cannot learn which tuples of the relation $S(R^1)_i$ have those tuple ids. Due to OP-SS, the adversary also knows only the order of values of $A_c$ attribute in the requested tuple ids. However, joining the tuple of $S(R^2)_i$, which has the maximum value in $A_c$ attribute, with all tuples of $S(R^1)_i$ 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 tuple ids (after round 1) and sends the desired tuple ids with some fake tuple ids, which do not satisfy the query predicate in the round 1, in cleartext to the servers. Let $x = r + f$ be the tuple ids that are transmitted to the servers, where $r$ and $f$ be the real and fake tuple ids, respectively. Note that the maximum value of the attribute $A_c$ over $x$ tuples may be more than the maximum value over $r$ tuples. Hence, the user does the following computation to appropriately send the tuple ids:

The user arranges the $x$ tuple ids in a $\sqrt{x} \times \sqrt{x}$ matrix, where all $r$ real tuple ids appear before $f$ fake tuple ids. Then, the user creates

---

*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.*
\[ \sqrt{\pi} \] groups of tuples ids, say \[ g_1, g_2, \ldots, g_{\sqrt{\pi}} \], 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 \( g_{max} \), contains both the real and fake tuple ids. Now, the user asks the server to find the maximum value of the attribute \( A_i \) in each group except for the group \( g_{max} \) and to fetch all \( \sqrt{\pi} \) tuples of the group \( g_{max} \).

**Server.** For each group, \( g_i \), except the group \( g_{max} \), each server \( i \) finds the maximum value of the attribute \( A_i \) by looking into the attribute \( CTID \) of the relation \( S(R^i) \) and results in a tuple, say \( \langle S(t_i), S(value) \rangle \). Further, the server \( i \) fetches all \( \sqrt{\pi} \) tuples of the group \( g_{max} \). 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 \( \square \) between all the tuples of \( S(R^1) \) and \( 2\sqrt{\pi} - 1 \) tuples obtained from the relation \( S(R^2) \), and returns \( 2\sqrt{\pi} - 1 \) tuples to the user. The user finds the maximum value over the \( r \) real tuples. Note that \( 2\sqrt{\pi} - 1 \) 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 tuple ids.** For finding the tuple ids, each server \( k \) executes the following operation: \( \text{Index}[i]_k \times (A_p[i] \oplus S(t_i)) \), 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 \( \text{Index} \) attribute of the relation \( S(R^1) \). Finally, the server sends all the \( n \) values of the attribute \( \text{Index} \) to the user, where \( n \) is the number of tuples in the relation. The user interpolates the received values and knows the desired tuple ids.

### 7.3 Verification of Maximum Query

This section provides a method to verify the tuple having maximum value in an attribute, \( A_i \). Note that verifying only the maximum value of the tuple is trivial, since \( \langle S(value) \rangle \) of \( S(R^i) \), is also a part of the attribute \( A_1 \) of \( S(R^1) \), and servers send a joined output of the relations (see step 2 in \( \square \)). Thus, servers cannot alter the maximum value. However, servers can alter other attribute values of the tuple. Thus, we provide a method to verify that the received tuple.

#### Verification of retrieved tuple

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

\[
\text{op}_1 = f_1(x) = \sum_{i=n}^{n} o_i(x_i + s_i) \\
\text{op}_2 = f_2(x) = \sum_{i=n}^{n} o_i(y_i + s_i)
\]

i.e., the server performs the functions \( f_1 \) and \( f_2 \) on some \( n \) values, (it will be clear soon what are these \( n \) values). In the above equations, \( o_i \) is the output of the string-matching operation carried on the \( i^{th} \) value of the \( \text{TID} \) attribute, and \( s_i \) 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 \( \text{Name} \) in Employee relation in Table \( \square \). The new value is the sum of the positions of each appeared alphabet in English letters; for example, the first value in the attribute \( \text{Name} \) is \( \text{John} \), the DB owner adds 47 (10+15+8+14). When creating shares of the two values at the \( i^{th} \) position of the attribute \( A_1 \) or \( A_2 \), the first value's shares are created using the mechanism that supports string-matching at the server, as mentioned in \( \square \) 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 \( \square \) numbers show newly added values to attributes \( \text{Name} \), \( \text{Dept} \), and random numbers (in bold-face) 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 \( \square \). 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 \( S(R^1) \). Finally, the server \( k \) sends the following two things to the user: (i) the tuple having the maximum value in the attribute \( A_i \) of the relation \( S(R^1) \); and (ii) the difference of outputs of the functions \( f_1 \) and \( f_2 \), say \( \text{diff}_{f1f2} \).

**User.** After interpolation, the user obtains the desired tuple and a value, say \( \text{Idiff}_{f1f2} \). 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 \( \text{Idiff}_{f1f2} \), and if it finds equal, the server has correctly sent the tuple.

**Example.** Table \( \square \) shows the verification process for the first tuple id of employee relation; see Table \( \square \). 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 OBSOURE and compares it against other SSS- and MPC-based systems. For Experiments 1-5, we used a single threaded implementation using AWS servers with 32GB RAM, 2.5GHz Intel Xeon CPU. Further, we used AWS servers with 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores (to evaluate the impact of parallelism using at most 32 threads, in Exp. 6). A 16GB RAM machine at the local-side worked as the DB owner, as well as, a user that communicates with AWS servers.

#### 8.1 OBSOURE Evaluation

**Secret-share (SS) dataset generation.** We used four columns (namely Orderkey (OK), Partkey (PK), Linenum (LN), and Supplier(SK)) of LineItem table of TPC-H benchmark to generate the dataset of 1M and 6M rows that were inserted into MySQL database system. 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 contains 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). 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, and it pre-

<table>
<thead>
<tr>
<th>Employee</th>
<th>Name</th>
<th>Salary</th>
<th>Dept</th>
<th>TID</th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>47</td>
<td>10000</td>
<td>80</td>
<td>3</td>
<td>1750</td>
<td>1235</td>
</tr>
<tr>
<td>106</td>
<td>47</td>
<td>100000</td>
<td>120</td>
<td>2</td>
<td>0</td>
<td>1000002012300</td>
</tr>
<tr>
<td>108</td>
<td>32</td>
<td>20000</td>
<td>31</td>
<td>4</td>
<td>0</td>
<td>1000002199500</td>
</tr>
<tr>
<td>109</td>
<td>30</td>
<td>1500</td>
<td>31</td>
<td>1</td>
<td>0</td>
<td>1000002195000</td>
</tr>
<tr>
<td>110</td>
<td>38</td>
<td>20000</td>
<td>31</td>
<td>6</td>
<td>0</td>
<td>1000002199500</td>
</tr>
</tbody>
</table>

Table 10: An execution of the tuple retrieval verification.
vents 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 (1 to 7), 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 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. 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 [2,1].

3. The third step was creating SS of these numbers. Thus, we selected a polynomial \( f(x) = \text{secret}_\text{value} + a_i x \), where \( a_i \) 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 \( S(R^1) \), and \( S(R^2) \), \( 1 \leq i \leq 15 \). (Exp. 4 will discuss in detail why we are generating fifteen shares.)

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 1 shows the time to generate SS LineItem table having four columns 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 OBSOURE is significantly less than an MPC system, which will be discussed in [2,1].

**Exp 2. Scalability of OBSOURE.** In order to validate the behavior of OBSOURE on a large-sized data, 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 also increases linearly, due to an increasing number of multiplications involved in string-matching operations.

**Count and sum queries.** Figure 2 shows the time taken by one-dimensional (1D), conjunctive-equality (CE), and disjunctive-equality (DE) 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 predicate increases, the computation time also increases, due to an increasing number of multiplications. Also, the time difference between computations on 1M and 6M rows is about 6-7%, which is proportional to increasing data size.

**Maximum queries.** Figure 3 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 \( S(R^1) \). Note that in both UnC-Max-Det and Cond-Max-Det, we compute 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%, which is also proportional to increasing data size.

**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. Figure 4 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. Overheads of result verification.** This experiment finds the overheads of the result verification approaches. Figure 24 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 25. Figure 25 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, in case of 1M and 6M rows, respectively.

**Exp 4. Impact of the number of shares.** Now, we discuss the impact of a different number of shares. For this experiment, we used 3, 5, 11, and 15 shares of the data. Due to space restriction, we show results of 1M rows. Figure 5 shows the computation time at the server and user, when having different shares, where black and white parts show the computation time at the user and server, respectively, and a bar shows the entire processing time.

The results demonstrate two tradeoffs: between the number of shares and computation time at a user, and between the number of shares and the amount of data transferred from a server to a user. As the number of shares decreases, the computation time at the user increases; since the string-matching operation results in the polynomial degree to be doubled, and if servers do not have enough shares, they cannot compute the final answer and may require more

<table>
<thead>
<tr>
<th>Tuples</th>
<th>Time</th>
<th>Size (in GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>10 min</td>
<td>(</td>
</tr>
<tr>
<td>6M</td>
<td>1.4 hours</td>
<td>(</td>
</tr>
</tbody>
</table>

Table 1: Exp. 1. Average time and size for shared data generation.
than one round of communication with the user to compute the SS aggregate value. Thus, the communication cost also increases.

From Figure 3, it is clear that as the number of shares increases, the computation time at user decreases and at the server increases, while the complete processing time also decreases, generally. For instance, consider CE-count on PK and LN (see Figure 3), where the number of digits in OK and LN were 6 and 1. For checking this predicate, we need at most 13 shares, as mentioned in [2]. When using 3 servers, having one of the three shares of the database, the computation time at each server and the user was 135s and 36s, respectively. In contrast, due to 15 servers, the computation time at each server and the user were 132s and some milliseconds.

A short reason is as follows: when we used 3, 5 or 11 shares, the computation time at the servers was almost same, since servers were only checking partial query predicates, i.e., digits of the number, and executing less number of multiplications, while the user was reducing the degree. In contrast, in case of 15 shares, servers checked the entire number and did more number of multiplications, while the user was executing Lagrange interpolation for obtaining the final answer. For our outsourced SS LineItem table, we can perform optimally all queries when the number of servers was 15, since all the predicates can be checked using 15 shares, except predicates involving three columns.

**Exp 5. 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 3) that was significantly more than the communication time between user and servers. Thus, the communication time does not affect the servers’ computation time, which was the bottleneck.

**Exp 6. Impact of multicore servers.** 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, DE count/sum, and unconditional maximum queries) that divide rows into blocks with one thread pro-

cessing one block, and then, the intermediate results (generated by each thread) are reduced by the master thread to get the final result. For this experiments having 15 shares, we used AWS servers with 144GB RAM, 3.0GHz Intel Xeon CPU with 72 cores, and varied the degree of parallelism (number of parallel threads) from 1 to 32. 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 (Amadahl’s law); furthermore, the execution time increases due to thread maintenance overheads. Figure 4 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.

**8.2 Comparing with Other Works**

The previous works on SSS-based techniques either did not report any experiments [27,23] or scaled to only a very small dataset, or used techniques that, while efficient, were insecure [28,45]. For instance, [28,45] are both vulnerable to access-pattern attacks. Furthermore, these approaches achieve efficient query processing times, e.g., 90 ms for aggregation queries on databases of size 150K by executing queries on SS data identically to that on cleartext, which requires user sides to retain polynomials, which were used to generate SS-data. Thus, as mentioned in [22], 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., [16,13,12], 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 50K 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 4. 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-theoretic secure and communication efficient aggregation queries (count, sum, and maximum having single dimensional, conjunctive, or disjunctive query predicates) on
a secret-shared dataset outsourced by either a single DB or multiple DB owners. We considered malicious adversarial cloud servers when all (or possibly the majority of) the servers deviate from the algorithm in an identical manner, due to software/hardware bugs. Thus, we proposed efficient result verification algorithms. Further, we evaluated our algorithms’ scalability and impact of parallelism.

10. REFERENCES


A. 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 tuples (A.1) and find the maximum value (or retrieve the tuple having the maximum value) over a dataset outsourced by multiple DB owner (A.2).

A.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 \( S(R^2) \), 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 \( S(R^2) \) and the attribute SS-\( A_c \) of \( S(R^1) \), 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 \( \langle S(t_k), S(value_1), S(value_2) \rangle \), having the maximum value (denoted by \( \langle S(value_1) \rangle \)) in the attribute \( A_c \), where \( S(t_k) \) is the \( k^{th} \) secret-shared tuple id (in the attribute SSTID) and \( \langle S(value_2) \rangle \) is the secret-shared value of the SS-\( A_c \) attribute in the \( k^{th} \) tuple. Afterward, the cloud \( i \) performs the following:

\[
\text{Index}[k] \times (A_c[S(k)]) \in S(value_2), 1 \leq k \leq n
\]

**i.e.,** the cloud compares \( S(value_2) \), 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 provide 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 \( T \), having the maximum value in the attribute \( A_c \):

\[
\sum_{k=m}^{k=n} A_p[S(a_k)], \times (\text{TID}[S(a_k)] \otimes S(t_j))
\]

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

**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 decreases 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 \( S(R^1) \) as suggested above. Instead of that the cloud provides a tuple having the maximum value in the attribute \( A_c \), and then, the user finds the 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 \( \text{index} \) 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.

### A.2 Finding Maximum over Datasets Outsourced by Multiple DB Owners

In this section, we explain a method, namedMDBMaxfor 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 \( j \) compares the signbit with all the other numbers using 2’s complement-based subtraction, i.e., \( \text{signbit}(V_i - V_j), x \neq i \), and \( 1 \leq x, i \leq n \). Recall that the signbit results in 1 of secret-share form, if \( V_i < V_j \); 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 \((3)\) 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 the cloud finds 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 signbit 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.

<table>
<thead>
<tr>
<th>( A_c )</th>
<th>Signbits</th>
<th>Sum of signbits</th>
<th>String-matching result</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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: \( \text{signbit}(V_i - V_j), x \neq i \), 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.

### B. MINIMUM, TOP-K, AND REVERSE TOP-K

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

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

**QMin1.** select \(*\), \( \text{MIN} \) (salary) from Employee

**QMin2.** select \(*\), \( \text{MIN} \) (salary) from Employee where E.Dept= ‘Testing’
1. On the relation $S(R^1)$, the user asks for the minimum value from the relation $S(R^1)$. First, each cloud $i$ finds a tuple, say $(S(tk_i), S(value)_i)$, where $S(tk_i)$ is the $k^{th}$ secret-shared tuple id (in the attribute $STID$) 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 $(S(tk_i))$ with each $k^{th}$ value of the attribute $TID$ of $S(R^1)$, 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 \[QMin\] 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 $S(R^1)$, and results in a tuple, say $(S(tk_i), S(value)_i)$, where $S(tk_i)$ shows the secret-shared tuple id (from $STID$ attribute) and $S(value)_i$ shows the secret-shared minimum value. Finally, the cloud $i$ performs a join operation between all the tuples of $S(R^1)$ and $(S(tk_i), S(value)_i)$, as performed when answering unconditional minimum ($QMin_1$) queries; see \[QMin\]. Correctness and information leakage of a minimum query is similar to maximum queries. Group-by. A group-by query finding the maximum/minimum can also be done in a similar manner as executing conditional maximum queries; see \[QMax\]. However, here the user asks for the tuple ids for each group in round 1 of the query execution, and then, asks to return a tuple having the maximum/minimum value in a specified attribute. Note that here the adversary again knows the number of groups and the number of tuples in each group. Hiding the number of tuples in a group can be done by following the method given in \[QMax\] which returns $2\sqrt{T} - 1$ tuples for a single group. Note that this method is only beneficial if the number of returned tuples for all the groups are significantly small than the number of tuples in the relation.

C. METHODS FOR FINDING TUPLE IDS

A trivial solution for knowing the tuple ids satisfying a query predicate is given in \[Q\] 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 $T$, satisfy the query predicate. We provide two methods: The first method executes $T$ computations on each tuples and maintains $T$ variables for each tuple. Thus, the cloud performs significant computations, when $T$ is large. The second method is only applicable when less than $\sqrt{n}$ number of tuples satisfy a query. This method maintains only three variable for each tuple.

The first method. The cloud creates $T$ columns\(^{10}\) 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 $T$ variables. For the purpose of explanation, we show $T$ columns. Each column has allocated one of the values from 1 to $T$ of secret-share form (provided by the user). After an oblivious computation over each tuple, if there are $T$ occurrences of $v$, then each of the $T$ columns will have one of the exact tuple id where $v$ occurs. The cloud executes the following operation:

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

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

Details. For $r^*$ (1 \leq r^* \leq n) value of the attribute $A_i$, the cloud executes counting operations for finding the occurrences of $v$ in $A_i$. The occurrences of $v$ in the above-equation is denoted by $a$. For

\(^{10}\)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.
each resultant $a$, the cloud compares $a$ against each of the $T$ values using 2's complement method (as given in [2]). The occurrence of $v$ matches with only one of the $T$ values, and thus, $\text{signbit}(x - a) + \text{signbit}(a - x)$ 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 $T$ 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 $T$, $\text{signbit}(x - a) + \text{signbit}(a - x)$ 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 $A_t$, the cloud keeps $r$ to one of the $T$ columns. It is important to note that if the $r^{th}$ tuple has $v$ in attribute $A_t$ and the $r^{th}$ tuple do not have $v$ in attribute $A_t$, the value of accumulated count, $a_t$, will be the same for the tuples $r^a$ and $r^{a+1}$. 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 $T$ column and sends the final sum of each column to the user.

Example. Table 12 shows an implementation of tuple id finding method in ciphertext 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 ($T = 2$) 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 $\text{signbit}(2 - 1) + \text{signbit}(1 - 2) = 0 + 1 = 1$. 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 $1 - (\text{signbit}(1 - 2) + \text{signbit}(2 - 1)) = 1 - (1 + 0)$. The cloud processes the remaining tuples in a similar manner.

<table>
<thead>
<tr>
<th>Tuple id</th>
<th>Dept</th>
<th>SM result ($o$)</th>
<th>Count ($a$)</th>
<th>$x = 1$</th>
<th>$x = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Testing</td>
<td>1</td>
<td>$r \times o = \text{signbit}(x - 1) + \text{signbit}(1 - x) = 3$</td>
<td>$r \times o = \text{signbit}(x - 1) + \text{signbit}(1 - x) = 0$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Security</td>
<td>0</td>
<td>$r \times o = \text{signbit}(x - 1) + \text{signbit}(1 - x) = 0$</td>
<td>$r \times o = \text{signbit}(x - 1) + \text{signbit}(1 - x) = 0$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Testing</td>
<td>1</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Design</td>
<td>0</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Design</td>
<td>0</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Design</td>
<td>0</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td>$r \times o = \text{signbit}(x - 2) + \text{signbit}(2 - x) = 0$</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Knowing tuple ids of employees working in testing department.

need to return the vector $(1, 1, 0, 0, 0, 0, 0)$ of secret-share form. The user can fetch the first and the second tuple obliviously be following the method given in [7,2]. 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 $M_X$. The $i$-th entry of $X$ now is denoted by $M_X[a,b]$, where $a$ is the quotient of $i$ divided by $\sqrt{n}$ and $b = i \mod \sqrt{n}$. The cloud performs an algorithm $\text{Compress}(M)$ (Algorithm 1), and the user performs $\text{Eval}(U, V)$ (Algorithm 2), where $M$ is a $\sqrt{n} \times \sqrt{n}$ and $U, V$ are vectors of length $\sqrt{n}$.

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

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

Algorithm 2: $\text{Eval}(U, V)$

1. Input: two vectors $U$ and $V$ of length $\sqrt{n}$.
2. Output: matrix $M$ which have 1 in the occurrences’ positions and 0 otherwise.
3. Initialize $M$ with 0;
4. for $i = 1$ to $\sqrt{n}$ do if $U[i] = 1$ then
5. 
6. $M[V[i], i] = 1$
7. Return $M$

Description: In our setting, all the data in clouds are in secret sharing form $S(*)$. However, the algorithm $\text{Eval}$ 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 $\text{Eval}$. Based on the definition of $\text{Eval}$, 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 $\text{Compress}(M^3)$, 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 $\text{Eval}$ can reveal rough positions for these occurrences, the third round communication is needed to reveal uncertain occurrences’ positions. Moreover, the communication cost

13Here the DB owner has to add one more column, say $\text{RowID}$ that has sequential numbers from 1 to $n$ in the relation $S(R^3)$, which helps in finding the required tuples.
in such a round depends on the number of occurrences. As mentioned, we assume that the number of occurrences for each pattern is lower than $\sqrt{\pi}$. In this case, we can recognize the matrix $M_X$ 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 $\text{Compress}(M_X)$ 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 $\text{Eval}(U_1, V_1)$.

**Round 2:** Clouds side: computes $\text{Compress}(M_X^T)$ 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 $\text{Eval}(U_2, V_2)$.

**Round 3:** After two rounds communication, the user will obtain the explicit positions for the occurrences which satisfy $\text{Eval}$. 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:

$$M_X = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}. \quad \text{(1)}$$

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

**Round 1:** Clouds compute $\text{Compress}(M_X)$, and set two vectors $U_1 = (0, 0, 3, 0, 3, 0, 1, 0, 0, 0, 0)$ and $V_1 = (0, 1, 2, 0, 1, 0, 0, 0, 0, 0, 0)$ to the user. The user checks $U_1, V_1$, only determines one occurrence’s position. Then, the user writes the matrix as follows:

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

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

**Round 2:** Clouds compute $\text{Compress}(M_X^T)$, and set two vectors $U_2 = (0, 0, 2, 0, 2, 0, 3, 0, 0, 0, 0)$ and $V_2 = (0, 6, 0, 6, 0, 12, 0, 0, 0, 0, 0)$ 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.

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

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.

### D. 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 [24]), 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, $q_{v_1}(D_1)$ and $q_{v_2}(D_2)$, respectively. Further, assume that $q_{v_1}(D_1)$ and $q_{v_2}(D_2)$ 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 $q_{v_1}(D_1)$ and $q_{v_2}(D_2)$ 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.