SEASEARCH: Secure and Efficient Selection Queries

Shantanu Sharma,[1] Yin Li,[2] Sharad Mehrotra,[3] Nisha Panwar,[4] Komal Kumari,[1] Swagnik Roychoudhury[5]
[1]New Jersey Institute of Technology, USA. [2]Dongguan University of Technology, China.
[3]University of California, Irvine, USA. [4]Augusta University, USA. [5]New York University, USA.

Abstract—Information-theoretic or unconditional security provides the highest level of security — independent of the computational capability of an adversary. Secret-sharing techniques achieve information-theoretic security by splitting a secret into multiple parts (called *shares*) and storing the shares across non-colluding servers. However, secret-sharing-based solutions suffer from high overheads due to multiple communication rounds among servers and/or information leakage due to access-patterns (*i.e.*, the identity of rows satisfying a query) and volume (*i.e.*, the number of rows satisfying a query).
We propose SEASEARCH, an information-theoretically secure approach that uses both additive and multiplicative secret-sharing, to efficiently support a large class of selection queries involving conjunctive, disjunctive, and range conditions. Two major contributions of SEASEARCH are: (*i*) a new search algorithm using additive shares based on fingerprints, which were developed for string-matching over cleartext; and (*ii*) two row retrieval algorithms: one is based on multiplicative shares and another is based on additive shares. SEASEARCH does not require communication among servers storing shares and does not reveal any information to an adversary based on access-patterns and volume.

Index Terms—Shamir's secret-sharing, additive secret-sharing, computation and data privacy, data and computation outsourcing, multi-party computation, selection query, conjunctive query, disjunctive query, range query

1 INTRODUCTION

This paper studies *information-theoretically secure* ways to support selection queries that may contain conjunctions, disjunctions, and range predicates. In contrast to encryption-based techniques that are only computationally secure (*i.e.*, secure against the adversary of limited computational capabilities), information-theoretically secure techniques offer a higher level of security. Such techniques remain secure regardless of the computational capabilities of an adversary (even with a quantum computer, at the present or the future) and are, thus, referred to as *unconditionally secure*. Secret-sharing (SS) is a popular information-theoretically secure technique. In a SS-based system, multiple pieces (called *secret-shares*) of a secret are created and placed into non-colluding (cloud) servers. To be able to reconstruct the secret, secret-shares from a number of servers (equal to or greater than a predefined threshold) need to be obtained. §1.1 will provide an overview of SS techniques. Advantages of information-theoretic security and the need of secure data outsourcing based on secret-sharing have been featured in several recent popular media articles [1]–[4].

While SS-based solutions require *multiple non-colluding servers*, with the emergence of several independent cloud vendors, such a requirement has become relatively easy to satisfy. Organizations already adopt multi-cloud solutions so as not to be locked into a single vendor for purposes such as fault-tolerance or vendor-specific dependency [5]–[9]. Organizations can further leverage multi-cloud settings to outsource secret-shares without concerns about cloud vendors colluding with each other to reconstruct user data.

Secret-sharing-based systems target single-table databases and support selection, aggregation, and group-by queries.[1] The demand for highly secure systems, even with limited operations, has driven multiple commercial solutions based on secret-sharing, *e*.g., Galois Inc.' Jana [10], [19], Stealth Software Technologies' Pulsar [11], and Cybernetica's Sharemind [12], [20]. Multiple systems to support single table queries using secret-sharing have also been developed by academia, *e*.g., Conclave [70], PDAS [69], and [34], [73]. Existing (academic and industrial) systems, however, suffer from the following two major drawbacks:

Information leakage. Existing systems do not prevent leakage due to access-patterns and/or volume, *simultaneously*. Access-pattern leakage refers to adversaries gaining knowledge of the identities of rows that satisfy a query, while volume or output-size leakage refers to the adversary getting to know the (output) size of query results. SS-based PDAS [69] and [34], [73] reveal both access-patterns and volume. Pulsar [11] and Conclave [70] reveal volume. However, *efficient* SS-based systems do not prevent both the leakages, simultaneously. Prior work has shown the importance of preventing both leakages [24], [43], [46], [51], [53], [54], [60], [65].

Query inefficiency. Systems that prevent both access-pattern and volume leakages simultaneously, are not efficient. For example, Jana [19] prevents both access-patterns and volume leakages, by returning the entire table with non-desired rows (*i.e.*, the rows that do not satisfy the selection query) being converted into zero of additive share form. However, Jana takes ≈450 s(econds) for selection queries on 1M rows. Another example is Sharemind [20], which also prevents both leakages, but it takes more than 600s for a simple projection query on 3M rows, as reported in [70]. The two main reasons for the query inefficiency are: (*i*) returning the entire data in the share form to prevent both access-pattern and volume leakages, while answering a selection query, and (*ii*) multiple rounds of communication among servers storing the shares to execute a selection query — particularly, for searching keywords over m columns and n rows can require $\mathcal{O}(m\ell)$ communication rounds where ℓ is the maximum length of a word, and the total

A preliminary version of this work was appeared in VLDB 2023.

1. While there are some works on multi-table join queries, such solutions are not practical. For instance, state-of-the-art solution for joins using secret-sharing described in [58] takes 2.6 seconds to join two tables with only 256 rows each!

Papers	[34]	S3ORAM [44]	Sepia [23]	Sharemind [20]	SPDZ [30]	Jana [19]	Conclave [70]	SEASEARCH
Technique	SSS	SSS	SSS	Additive	Additive	Additive	Additive	**Additive + SSS**
Communication between servers	No	Yes	Yes	Yes	Yes	Yes	Yes	**No**
Distribution/frequency leakage from ciphertext	No	No	No	No	No	No	No	**No**
Access-pattern leakage from query execution	Yes	No	No	No	No	No	Yes	**No**
Volume leakage from query execution	Yes	No	No	No	No	No	Yes	**No**
Supported operators	SPJ	1 keyword fetch	Compare/ equality	SPJ	SPJ	Selection	†	**Complex search & fetch**
Computational complexity of selection	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	N/A	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\boldsymbol{\mathcal{O}(n)}$
Index support	Yes	Yes	No	No	No	No	No	**No**
Support for dynamic data	Yes	No	N/A	Yes	Yes	Yes	Yes	**Yes**
Experimental results: Time *using one thread* (data)	**0.07s∗ (150K rows)**	**7.3s (on 40GB)‡‡**	**N/A**	**>10m (on 3M rows)◇**	**10s (on 1000 rows) ⊖**	**450s (on 1M rows)**	†	**1.502s on 1M rows**

TABLE 1: Comparison of existing secure systems against SEASEARCH. Notes. (*i*) SSS: Shamir's secret-shares. (*ii*) SPJ: selection, projection, join. (*iii*) s: seconds. m: minutes M: Millions. (*iv*) The scalability numbers are taken from the respective papers. (*v*) ∗: Numbers are taken from [73] experimental comparison. (*vi*) ‡‡: Does not mention the number of rows. (*vii*) ◇: Numbers are taken from Conclave [70] experimental comparison. (*viii*) ⊖: Numbers from [56]. (*ix*) †: **Conclave [70] uses a trusted party** to support SPJ over multi-party settings, and thus, we do not include experimental numbers.

amount of information flow among servers (and between a server and a querier) can be $\mathcal{O}(nm)$.

This paper describes ***efficient, scalable, and information-theoretically secure techniques for selection queries***, entitled SEASEARCH,[2] that prevents both information leakages from access-patterns and volume. Also, SEASEARCH does not require servers (which store secret-shares) to communicate among themselves before/during/after computations. SEASEARCH offers: (*i*) ***query privacy*** — indistinguishability of queries by an adversarial server, (*ii*) ***data privacy*** — not revealing to an adversarial server anything (such as data distribution and ordering) about input/intermediate/output data, and (*iii*) ***server privacy*** — not revealing to the querier/client anything other than the answer to the query.

Before describing, how SEASEARCH achieves the goal of efficient, scalable, and secure search, we first briefly discuss secret-sharing techniques.

1.1 Background on Secret-Sharing & Fingerprints

SEASEARCH uses additive shares, multiplicative shares, and fingerprints.

Additive Secret-Sharing: is the simplest type of secret-sharing method. Additive shares are defined over an Abelian group, $\mathbb{G}_p$, under addition operation modulo p, where p is a prime number. A secret owner creates $c{>}1$ shares of a secret, say S, such that $S{=}\sum_{i=1}^{i=c} s_i$ (s_i denotes an i^{th} share) over $\mathbb{G}_p$, and sends s_i to the i^{th} server (belonging to a set of c non-colluding servers). These servers cannot know S unless collecting all c shares. To reconstruct S, the secret owner collects all c shares and adds them.
Example. Let $\mathbb{G}_5{=}\{0,1,2,3,4\}$ be an Abelian group under addition modulo 5. Let 4 be a secret. A secret owner may create two shares: 3 and 1 (since $4{=}(3{+}1) \bmod 5$) and send them to two servers.
Property. Additive shares of a number are random (*e.g.*, $5{=}1{+}4$ and $5{=}2{+}3$); thus, the adversary by observing an additive share cannot deduce a secret. Additive sharing allows *additive homomorphism* (*i.e.*, adding two or more shares at a server *locally*, *i.e.*, without communicating with other servers) and *scalar multiplication* (*i.e.*, multiplying a number to all additive shares, and the result is equivalent to multiplying two numbers in cleartext).

Multiplicative Secret-Sharing. The classical multiplicative secret-sharing scheme was introduced by Adi Shamir [67], say Shamir's secret-sharing (SSS). It requires a secret owner to randomly select a polynomial of degree c' with c' random coefficients, *i.e.*, $f(x) = a_0{+}a_1x{+}a_2x^2{+}\cdots + a_{c'}x^{c'}$, where $f(x){\in}\mathbb{F}_p[x]$, p is a prime number, $\mathbb{F}_p$ is a finite field of order p, $a_0{=}S$ (the secret), and $a_i{\in}\mathbb{N}$ ($1{\leq}i{\leq}c'$). The secret owner distributes S into $c{>}c'$ shares, by computing $f(x)$ for $x{=}1,2,\ldots,c$ and sends an i^{th} share to the i^{th} server. The secret, S, is reconstructed using Lagrange interpolation [28] over any $c'{+}1$ shares. An adversary can construct S, iff they collude with $c'{+}1$ servers. Thus, the degree of a polynomial is set to be c', if an adversary can collude with at most c' servers. In this paper, the terms 'multiplicative secret-sharing' and 'Shamir's secret-sharing (SSS)' are used interchangeably.
Property. SSS allows *additive homomorphism*. SSS offers *multiplicative homomorphism*, *i.e.*, servers can locally multiply shares, and the result can be constructed at the owner if we have enough shares, as multiplication increases the polynomials' degree.

Fingerprint. The fingerprint (function) was proposed for string matching over cleartext [48]. A fingerprint is defined as: $\phi_{r,p}(s) = \sum_{i=1}^{l} s_i r^i \bmod p$, where a string $s = s_1 s_2 \ldots s_l$ coded over a finite field $\mathbb{F}_p$, p is a prime number, and $r \in \mathbb{F}_p$ is a random number.
Property. Fingerprint functions are additive homomorphic, *i.e.*, $\phi_{r,p}(s_1{+}s_2) = \sum_{i=1}^{l}(s_{1,i}{+}s_{2,i})r^i \bmod p = \phi_{r,p}(S_1){+}\phi_{r,p}(S_2)$. Two identical strings always generate the same fingerprint. **SEASEARCH *executes fingerprints over additive shares for string search and conjunctive search*** (see §5.1.1). As fingerprints execute a modular operation, the probability of false positives (*i.e.*, answering a string even if it does not match against a pattern) exists. Nonetheless, the probability of false positives is very small, *i.e.*, l/p, with l being the string's length and p ($\gg l$) being a large prime number. False positives exist, usually, for random strings. If we consider only meaningful strings (like names of people), the error probability is negligible. More details may be found in §2 of [22]. In our experiments with $r = 43$ and $p = 100{,}000{,}007$, we get zero false positives for any query. The readers must not relate the fingerprint false positive probability against the birthday

2. Finding items in a sea is practically impossible if the exact location of items is unknown; likewise, an adversary not knowing all shares cannot learn the secret and cannot execute a query without knowing a sufficient number of shares.

Notations	Meaning
R and A	A relation/table R in cleartext and the attribute/column A of R
$\mathbb{R}$	A relation/table in share form
$\mathbb{A}$	A column A of R in additive share form
$\mathbb{A}.v_i$	A value in additive share in $\mathbb{A}$ in an i^{th} row
$\mathfrak{A}.v_i$	A value in multiplicative share in $\mathbb{A}$ in an i^{th} row
n and m	The number of rows and the number of columns in the table R
$\mathcal{S}_i$ and $\mathcal{S}_c$	A server i and the combiner
$F(x)$ and r	A fingerprint of x using the fingerprint parameter (a prime number) r
p	A prime number used as modulo in secret-sharing
$\mathcal{PRG}$ and $seed$	A pseudo-random generator and the seed used in $\mathcal{PRG}$

TABLE 2: Frequently used notations in this paper.

paradox [49], and in fingerprint functions, p is a modulus, not the range domain used in the birthday paradox.

1.2 Summary of SEASEARCH

SEASEARCH, primarily, ***supports selection queries containing conjunctive, disjunctive, and range predicates. Also, SEASEARCH can offer sum and group-by sum queries.***

To execute a selection query, SEASEARCH executes two rounds of communication between servers and a client. In the first round, SEASEARCH finds the row ids that satisfy the query predicate, and then in the second round, fetches all the qualified rows. SEASEARCH uses additive shares for single/multiple keyword search, conjunctive search, and search involving range conditions, while uses multiplicative shares for disjunctive search. Importantly, SEASEARCH *does not require servers (which store secret-shares) to communicate among themselves before/during/after computations*. All supported operations by SEASEARCH prevent both access-patterns and volume leakage, simultaneously.

Each round of SEASEARCH comes with a challenge (discussed below). Also, we provide an overview of the solution to address the challenge.

1. Efficient and Oblivious Search. The challenge in round one of SEASEARCH is ***to search query keywords efficiently over one or multiple columns (i.e., conjunctive and disjunctive search) obliviously*** (*i.e.*, being data-independent and not revealing access-patterns, as well as volume). In simple words, the search operation supported by SEASEARCH at the cloud must be completely private. A trivial and impractical approach to address this challenge is to download a complete copy of the entire data and then execute the query locally. Another straightforward solution is to use a keyword search protocol such as [37] or private information retrieval (PIR) by keywords [26]. However, in the case of keyword search protocol [37], the query size and computation cost at a server will be equal to all possible combinations of unique keywords across all columns of a database (see §2.3 of [57]). In contrast, PIR by keyword will reveal additional data to a client, *i.e.*, the client will not only learn the desired data, but also learn other data without executing queries for them, (see §1.1 and §4.2 of [37]).

Our approach. To address the problem of efficient and oblivious search, we develop novel search techniques using fingerprint-based search [48], which was developed for string matching over cleartext. Our search algorithm uses the concept of fingerprints and enables them to work over additive shares. The novelty of the algorithm is that it ***does not require communication among servers to perform a search operation over one or more columns, due to utilizing additive homomorphism of both fingerprints and additive shares***. The search algorithm takes as inputs keyword(s) in secret-share form and outputs the row-ids, where the keyword appears in the secret-share table. The search algorithms need only one round of communication between the server and the client. In terms of security, the search algorithms do not reveal access-patterns and volume to servers.

Threads	String search	Number search	Conjunctive search	Disjunctive search	Row Fetch — Multiplicative	Row Fetch — Additive
1 thread	**0.783**	**0.582**	**0.696**	**0.743**	**0.759**	**0.950**
4 threads	**0.453**	**0.396**	**0.451**	**0.475**	**0.395**	**0.452**

TABLE 3: **SEASEARCH** performance (in sec) on 1M rows using 1 and 4 threads.

2. Efficient and Oblivious Row Retrieval. Once we know the row ids that have the query keyword in round one of SEASEARCH, the next challenge is ***to fetch the row without revealing to servers access-patterns and volume***. To address this, one possible solution is to use oblivious random access memory (ORAM) [40], [68]. However, ORAM schemes have multiple drawbacks: revealing additional data other than the answers to the query to the client, no support for queries with conjunctive/disjunctive conditions, no support for range queries, no efficient support for dynamic data, and harder to support multiple clients, as also argued in [32]. Another possible solution is to use PIR or optimized versions of PIR, known as Distributed Point Function (DPF) [38] and Function Secret Sharing (FSS) [21]. SEASEARCH provides two methods to fetch rows, and one of them is built on DPF.

Our approach. We develop two methods: the first method (§6.1) uses multiplicative shares and incurs the communication cost of $\mathcal{O}(\sqrt{n})$ from a client to a server, where n is the number of rows in a table. The second method (§6.2) uses additive shares and incurs the communication cost of $\mathcal{O}(\log n)$ from a client to a server. The second method is inspired by DPF and leverages DPF to fetch additive shares obliviously. Both methods hide access-patterns when fetching rows, only utilize four servers, and do not need servers to communicate among themselves during the protocol. Moreover, both methods are designed to fetch $\mathcal{O}(\sqrt{n})$ rows in the same round with the same communication and computational cost. Importantly, our methods are significantly better than existing work [19], [20], [52] that requires each server to send the entire table containing the desired rows (*i.e.*, the rows satisfying the query predicates) and non-desired rows being converted into zero in the share form. Table 1 compares current secret-sharing-based schemes and SEASEARCH.

3. SEASEARCH performance. We implemented SEASEARCH in Java, and the code contains more than 9,000 lines. We set up SEASEARCH at AWS and tested on 1M and 10M rows of Lineitem Table of TPCH benchmark [13]. In *round one for a search query* (*i.e.*, knowing the row ids), SEASEARCH took at most 0.475s on 1M rows and 2.964s on 10M rows using four-threaded implementation. In *round two to fetch row*, SEASEARCH took 0.395s using multiplicative shares and 0.452s using additive shares on 1M rows using four-threads. On 10M rows, in round two, SEASEARCH took 6.765s using multiplicative shares and 3.357s using additive shares using four threads. Table 3 provides computation time over 1M rows using one/four threads.

Method	Total query time	Speedup	Operation support
DL encrypted data	14.6s(1.6s to DL, 10.8s to decrypt, 2s to load, .2s to Q)	$\approx 10x$	Any
DL additive shares	4.9s (1.2s to DL, 1.5s to add, 2s to L, 0.2s to Q)	$\approx 3x$	Any
DL one-time pad	4.6s (1.2s to DL 1.2s to XOR 2.0s to load 0.2s to Q)	$\approx 3x$	Any
Jana	$\approx$450s	$\approx 300x$	Selection query
Waldo	$\approx$12s	$\approx 7x$	Absence/presence of a keyword
SEASEARCH	**$\approx$1.5s (0.743s for search, 0.759 for row fetch)**	x	**Selection query**
Notations: DL: Download. L: Load data into MySQL, Q: query execution without index.			

TABLE 4: Comparing different systems against SEASEARCH on 1M rows.

Entities	Parameters
DBO	All parameters except $seed_c$ (*i.e.*, the seed for $\mathcal{PRG}$ selected by the client)
Server $\mathcal{S}_i$	p, r, F, $\mathcal{PRG}$, $seed_S$ (the seed only known to the servers), $seed_c$
Client	p, r, F, $\mathcal{PRG}$, $seed_c$
Combiner $\mathcal{S}_c$	p

TABLE 5: Parameters known to different entities.

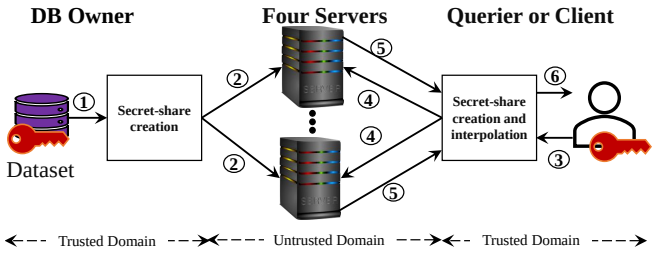


Fig. 1: The model.

4. SEASEARCH vs other approaches, and the reason for using fingerprints. The following multiple simpler approaches could support the same class of operators as SEASEARCH: (*i*) *Downloading the entire encrypted data:* the client downloads the entire encrypted data, decrypts it, loads the cleartext data into a DBMS, and then executes a query. (*ii*) *Downloading all the additive shares:* the client downloads additive shares of the data, performs additions over the shares to obtain the cleartext data, and then loads it into a DBMS to execute the query. (*iii*) *Using one-time pad:* one server stores the one-time pad [50] and another server stores the XOR of the pad with the database, and for query processing, the client fetches the data from both servers to recompute the original database and executes a query after loading the data into a DBMS.

While the first approach is not information-theoretically secure, the remaining two are. The second and the third approaches offer the same security as SEASEARCH offers, do not require dependence on fingerprints, and do not require any communication among the cloud servers.

All three approaches, however, incur a huge communication cost in downloading the data and the computational cost at the client to execute a query. Compared to these three approaches, SEASEARCH first finds the desired row ids, and then, fetches those rows. It is important to note, a fingerprint function, in SEASEARCH, compresses the output of string comparison, as well as, compresses the output of a conjunctive search over $x > 1$ columns and returns only a single value/number per row regardless of x columns. Table 4 compares SEASEARCH against the above three approaches and also to other approaches.

Code. Code and data used in experiments for this paper are available at [14].

2 PRELIMINARY

This section provides an overview entities involved in SEASEARCH, the threat model, and the security properties.

2.1 Entities and Assumptions

We assume three entities (database owner, servers, and clients/queriers); see Figure 1. Table 2 provides frequently used notations in the paper. Table 5 shows parameters known to different entities.

1) **Database owner (DBO)** owns a database and creates both additive and multiplicative secret-shares of the database (§4 provides the algorithm for creating secret-shares). DBO transfers the i^{th} share of the database to the i^{th} server.

2) **Servers** store secret-shared data outsourced by DBO and execute queries for clients. The servers are untrusted. The security model requires that the *secret-shared data stored at the server must not reveal anything about the data*, *e.g.*, data distribution and ordering of values, to the server. Likewise, *query answering protocols must not reveal anything about the client's query*, *e.g.*, the query, answers to queries, access-patterns, and volume, to the server. As will become clear soon, we use four servers $\mathcal{S}_{z\in\{1,4\}}$ to store secret-shares, where $\mathcal{S}_1$, $\mathcal{S}_3$ store the same shares and $\mathcal{S}_2$, $\mathcal{S}_4$ also store the same shares.

   For developing our technique, we make a simplifying assumption that servers do not collude with each other. Since our technique is based on secret-sharing, this assumption can be relaxed by increasing the number of shares. Note that secret-sharing is robust against collusion amongst servers and byzantine behavior as long a the majority of the servers are not malicious, *i.e.*, do not collude and follow the protocol correctly [29], [36], [62]. Servers establish a secure communication link with DBO and clients, and authenticate them before executing the protocols.

3) **Queriers/Clients** ask queries over secret-shared data, stored at the servers. Clients, in general, can be different from DBO. While DBO can be a client, a client might not be the owner of the database. If a client is different from DBO, the client's access to data is restricted based on policies specified by DBO. Restricting client's access requires an additional access control

mechanism to ensure that client's queries are restricted to data for which the client has access permission. Standard access control mechanisms (*e.g.*, as [59]) can be used for such a purpose. We will, henceforth, assume the presence of such a mechanism and, thus, restrict our protocols to the case when clients' queries only access data they have permission to.

Our security model requires *the protocol to guarantee that the client only learns answers to the query and nothing else*, *i.e.*, clients does not learn any information (*e.g.*, distribution/ordering) about data which it does not query. Since the client's queries do not access data the client does not have access permission for, and, furthermore, since our protocol guarantees that the client learns nothing about data other than the data it queries, our protocols guarantee that the client does not gain any information about data they do not have access to. We develop our protocol under the assumption that the server does not collude with DBO (if indeed DBO is different from a querier) to identify the query being asked by the querier.

2.2 Security Requirements and Properties

We follow the standard security definition given in [18], [37]. Particularly, we need to satisfy the following three specific properties:

Query privacy requires that the queries or query predicates must be hidden from servers, and they cannot distinguish between arbitrary queries or query predicates. For example, queries searching over a column for keywords must be indistinguishable by servers.[3]

Data privacy requires that the stored input data, intermediate data during computation, and output data are not revealed in cleartext to servers, and the secret value can only be reconstructed by the client. We must ensure that the servers will not learn (*i*) frequency distribution, *i.e.*, the number of ciphertext rows containing an identical value, (*ii*) ordering of values, *i.e.*, a relationship of $<, >, =$ between two shares, (*iii*) access-patterns, *i.e.*, the identity of ciphertext satisfies a query, and (*iv*) output-size/volume, *i.e.*, the number of ciphertext satisfies a query. Based on such things, an adversary may learn the full/partial data, as discussed in [24], [46], [51].

Server privacy restricts a client from learning additional information other than the answers to the queries [57]. Server privacy is important when the queriers are different from DBO.[4]

Now, to formally define the security properties, we define the notion of adversarial view, below.

Adversarial view and notations for security definition. An adversarial server knows secret-shared data (n rows and m columns), secret-shared query predicates, and the protocol they execute. This is known as *adversarial view*. The adversarial view *does not* captures the cleartext data, cleartext query predicates, and cleartext answers to the queries. Based on the adversarial view, the adversarial server wishes to learn the cleartext data, query, and/or results. (Recall that a majority of the servers cannot collude with each other; thus, servers cannot reconstruct the cleartext.)

3. This paper does not focus on hiding the type of queries (*i.e.*, conjunctive, disjunctive, range query, or fetch queries). Hiding the query type will lead to huge redundant data, which eventually incurs computational overhead.

4. Consider a hospital database on which a nurse is executing a query to know all rows of cancer patients. Now, here, due to the query execution, the nurse must not learn anything about other diseases, such as data distribution of other diseases or which sensitive disease is not treated by the hospital. In this case, server privacy plays an important role in preventing this.

Let $\mathcal{A}_{view}(\pi, qp, \mathbb{R}, \mathcal{L})$ be the adversarial view of an adversary $\mathcal{A}$ in the real execution of a protocol π (for a query predicate qp) on input secret-share table $\mathbb{R}$ of $n{>}1$ rows and $m{>}0$ columns. Here, $\mathcal{L}$ refers to the rows accessed by the protocol π, *i.e.*, access-patterns, and the size of outputs, *i.e.*, volume. The protocol π is executed on more than one server that stores secret-share table $\mathbb{R}$. Note that in SEASEARCH, a protocol π could be either any type of search or row fetch, and qp could be any keyword for search protocols or row-ids for row fetch protocols. The security of a protocol is defined as follows:

Definition 1. *For an adversary $\mathcal{A}$ executing a server protocol π over any input secret-share relation $\mathbb{R}$ of $n{>}1$ rows and $m{>}0$ columns and for any query predicates qp, qp', π is secure, iff the following condition holds (where $\mathcal{L}$ is defined above):*

$$\mathcal{A}_{view}(\pi, qp, \mathbb{R}, \mathcal{L}) = \mathcal{A}_{view}(\pi, qp', \mathbb{R}, \mathcal{L}) \blacksquare$$

Definition 1 indicates that the *adversary cannot distinguish two (or more) adversarial views* obtained by executing the same protocol for two (or more) query predicates. Thus, the adversary cannot learn anything based on secret-shared tables and/or query execution, such as frequency distribution, ordering, access-patterns, and volume. Particularly, the adversary cannot distinguish (*i*) which query predicates they are working on, and (*ii*) which rows and how many rows of the table satisfy the query. Thus, the protocol satisfies the properties of query privacy and data privacy.

Definition 2. *For any given secret-shared relation at the servers, for any query predicate qp, and for any real client, say C, there exists a probabilistic polynomial time (PPT) client C' in the* ideal execution*, such that the outputs to C and C' for the query predicate qp on the secret-shared relation are identical.* $\blacksquare$

Definition 2 indicates that no client will learn more data other than the answer to a query predicate qp. The security proof of the definitions is provided in the Appendix.

Later sections will discuss in detail the security of different operators offered by SEASEARCH. Informally, SEASEARCH uses secret-sharing techniques over cleartext to produce non-identical shares for the same value and non-orderable shares for values holding an order $(>, <, =)$ in cleartext. This prevents frequency distribution and ordering leakages from secret-shared data. SEASEARCH sends query predicates in share form to prevent adversaries from learning the query predicates. Queries are executed obliviously to prevent leakages from access-patterns. Further, SEASEARCH returns the same amount of output for any search (either over one/multiple columns or conjunctive/disjunctive search) and for row retrieval to hide volume. Also, servers return data in a way that only reveals answers to the queries, nothing else, to the client.

2.3 Evaluation Parameters

SEASEARCH can be evaluated on the following theoretical parameters: (*i*) *Computation cost*: is measured at a client and a server, and finds the number of values on which each entity performs computation for answering a query. (*ii*) *Scan cost*: finds the number of rounds, when a server and a client read n values. (*iii*) *Communication rounds and cost*: are measured between a client and a server. The communication round is the number of times data flows between a client and a server to execute a query. Communication cost finds the amount of data flowing between a client and a server. The ***communication cost among servers is***

Operations	Shares used	Server used
Search (including single- or multi-keyword, conjunctive, range)	Additive	$\mathcal{S}_1, \mathcal{S}_2$
Disjunctive search	Multiplicative	$\mathcal{S}_1, \ldots, \mathcal{S}_4$ (if >3 disjuncts)
Fetching a row	Multiplicative or additive	$\mathcal{S}_1, \ldots, \mathcal{S}_4$
Group-by count	Additive shares	$\mathcal{S}_1, \mathcal{S}_2$
Group-by sum	Additive shares	$\mathcal{S}_1, \ldots, \mathcal{S}_4$

TABLE 6: Techniques and servers used for different operations.

always zero, since SEASEARCH does not require communication among servers.

Briefly, in SEASEARCH, each operator takes only one round of communication between the desired two entities. The maximum communication cost from a server to a client is $\mathcal{O}(n)$ bytes (depending on the size of integers used in programming languages), where n is the number of rows, while from a client to a server is $\mathcal{O}(\sqrt{n})$. The scan cost at servers and a client is one. The computation cost varies for different operations and is discussed with each operator.

3 SEASEARCH AT THE HIGH LEVEL

SEASEARCH consists of three entities: a trusted DBO, four untrusted servers, and clients; see Figure 1. SEASEARCH provides *oblivious* algorithms for search and row retrieval. At the abstract level, data processing in SEASEARCH consists of the following four phases:

First phase: Data outsourcing ①②. DBO creates additive and multiplicative shares of the data (using a method given in §4). For strings, DBO first converts them into a sequence of numbers by translating each letter to a number (*e.g.*, according to the position in a language), and then, additive and multiplicative shares are created for such numeric strings. For numbers, DBO simply creates both types of shares. All shares are outsourced to servers.

Second phase: Secret-sharing creation of queries ③④. Queries are initiated by a client by creating secret-shares of query predicates, which are sent to servers.

Third phase: Search query execution. Servers execute the algorithm *locally*, depending on the requested search operation. Particularly, servers execute computations over additive shares for a single keyword, multi-keyword, conjunctive, and range search, while multiplicative shares are utilized for disjunctive search. On completing the algorithm, each server sends a vector ⑤ in share form to the client. The algorithm's execution does not reveal access-patterns and volume, as well as query predicates/answers/input to servers.

Final phase: Fetch operation. The client determines the final answer of a search query, *i.e.*, which row-ids contain the query predicate, by interpolating the received vectors ⑥. If the client wishes to fetch the rows also, the client communicates one more time with the four servers. Then, servers, *locally*, execute either multiplicative- or additive sharing-based method that obliviously returns rows to the client, which interpolates the received shares and obtains the rows.

rid	Name	Cost
1	Jo	4
2	Mo	6
3	Lo	8
4	Mo	4

TABLE 7: An input cleartext `Patient` table.

rid	A.NAME	A.COST	M.COST
1	6,10	3	6
2	10,5	2	8
3	10,6	4	10
4	3,5	2	6

TABLE 8: PATIENT$_1$.

rid	A.NAME	A.COST	M.COST
1	4,5	1	8
2	3,10	4	10
3	2,9	4	12
4	10,10	2	8

TABLE 9: PATIENT$_2$.

4 DATA OUTSOURCING IN SEASEARCH

This section explains how SEASEARCH uses different secret-sharing techniques on a table/relation and outsources them. To state it briefly, SEASEARCH *creates both additive and multiplicative shares of each value*. A summary of such techniques and servers used for different operations is given in Table 6. Detailed reasons of using different types of shares will be clear soon. The method for share creation is explained using a `Patient` table; see Table 7.

C1: Shares for strings. We maintain a mapping for each letter to a number. This mapping could be either the position of the letter in the language or the ASCII code. First, each letter in a string is converted into a number according to the mapping of the letter. Second, since strings can be of different lengths, which may reveal information to the adversary, an identical random number is padded to strings to make them equal in length. Finally, two additive shares ($\mathbb{A}.v^1$ and $\mathbb{A}.v^2$) of each number v are created. When the meaning is clear, $\mathbb{A}.v$ instead of $\mathbb{A}.v^1$ or $\mathbb{A}.v^2$ will be used. Also, two multiplicative shares of v are created. Recall that in §2.1, for the purpose of simplicity, we assume that servers do not collude with others; thus, for multiplicative shares, polynomials of degree are enough.

C2: Shares for numbers. We create additive and multiplicative shares of each number.

Data outsourcing. On each column of a table R, DBO implements the above method, depending on the column containing strings or numbers. This produces two tables $\mathbb{R}_1$, $\mathbb{R}_2$, where $\mathbb{R}_i$ contains the i^{th} shares. ***DBO outsources*** $\mathbb{R}_1$ ***to servers*** $\mathcal{S}_1$,$\mathcal{S}_3$ ***and*** $\mathbb{R}_2$ ***to*** $\mathcal{S}_2$,$\mathcal{S}_4$.

Aside. The detailed reasons of using four servers will be clear in §5.3,§6. In short, the two additional servers are used only in disjunctive search or fetching a row. Recall that one of our row fetch methods is based on multiplicative shares, and this method performs two times multiplications over shares with another multiplicative share, which were created from polynomials of degree one. Thus, four servers allow the client to interpolate polynomials of degree three. Another row fetch method is based on additive shares and is based on DPF. DPF works over two servers, and both servers keep identical data in cleartext. To make DPF work for additive shares, we use four servers and replicate a share over two servers.

Example. Table 7 shows a cleartext table, whose secret-share tables using the above method are shown in Tables 8, 9. To

illustrate, we use rid column to refer to row-ids, but this column is not needed to outsource. Consider a prime number $p = 17$. DBO wants to create shares of "**Jo**" and "**4**" (name and cost values in the first row of Table 7). To do so, DBO represents Jo by letters' positions: $\langle 10, 15\rangle$, and then, creates two additive shares of $\langle 10, 15\rangle$, as: $\langle 6, 10\rangle$ and $\langle 4, 5\rangle$. We do not show multiplicative shares of the name column. The first share table $\texttt{Patient}_1$ contains $\langle 6, 10\rangle$ in the name column, while $\langle 4, 5\rangle$ is kept in the name column of the second share table $\texttt{Patient}_2$. DBO creates $\langle 3, 1\rangle$ as the additive shares of 4. For creating multiplicative shares of 4, DBO uses a polynomial of degree one (*e.g.*, $f(x) = (2x + s) \bmod p$, where $s = 4$ is the secret value) and obtains shares as: $f(1) = 6$ and $f(2) = 8$. *For simplicity, only one polynomial for the entire Table 7 is selected.* ■

Discussion on leakages from the secret-shared data. The common leakages from ciphertext may reveal the frequency distribution and ordering of the values. Our share creation method prevents both of these leakages. Particularly, additive and multiplicative shares of a value are randomly created (resulting in non-identical shares). Thus, by observing shares, an adversary cannot deduce whether two or more shares correspond to an identical value or hold any relation $(<, >, =)$, preventing an adversary from learning frequency distribution and ordering information from the shares. Also, note that while DBO adds an identical random number to make strings of the same length, the adversary cannot deduce which share corresponds to a real or fake number, due to randomness in creating shares.

5 Keyword Search Algorithms

This section develops operators to search keywords over a single or multiple columns: single keyword search in a column (§5.1), conjunctive search (§5.2), and disjunctive search (§5.3). These operators involve servers and a client and facilitate the client knowing the row-ids that satisfy a query. §5.4 provides methods to optimize the communication cost between servers and clients for practical usage.

5.1 Single Keyword Search

A single/simple search operator finds whether a keyword/query predicate exists in secret-shared data or not.

5.1.1 *High-level Idea and Step-wise Details*

The idea of our search operator is that if we subtract two identical strings that are represented as numbers according to their letters' positions (in the language), then the result will be zero; otherwise, a non-zero number. SeaSearch implements exactly the same idea over additive secret-shares and query predicate at two servers.

Fingerprints are used to compress the string-matching outputs and a pseudo-random generator (PRG) is used to provide security. Below, we explain how the search algorithm works over strings:

1) ***Client***: represents the query keyword according to their positions in English alphabets, creates two additive shares of the keyword (as explained in C1 in §4), and computes fingerprints over secret-shares. Fingerprints for a query keyword q are denoted as $F(q_1)$ and $F(q_2)$, and $F(q_z)_{z\in\{1,2\}}$ is sent to server $\mathcal{S}_z$.
   Also, the client negotiates $\mathcal{PRG}(seed_c)$ with $\mathcal{S}_1$. Note that in this protocol between the client and servers, such a PRG is not necessary. ***The reason for using $\mathcal{PRG}(seed_c)$ is to only reduce the total communication between servers and client at the cost of an additional new untrusted server***, and will be clear in §5.4.

2) ***Server***: $\mathcal{S}_{z\in\{1,2\}}$ executes three operations: (*i*) computes fingerprints over the data, (*ii*) subtracts the fingerprint received from the client, and (*iii*) multiplies and adds random numbers. Particularly, $\mathcal{S}_z$ computes fingerprints over additive shares of the desired column, $\mathbb{A}$, and subtracts the received fingerprints $F(q_z)$ from each fingerprint in $\mathbb{A}$. Then, $\mathcal{S}_z$ multiplies a random number, *i.e.*, $\mathcal{PRG}(seed_s)$ and also adds a random number, *i.e.*, $\mathcal{PRG}(seed_c)$, where $seed_s$ is unknown to clients and $seed_c$ is known to $\mathcal{S}_1$ and clients. $\mathcal{PRG}(seed_s)$ ***is added to achieve* server privacy**, will be discussed in §5.1.3. Particularly, for each j^{th} row,

$$\boldsymbol{\mathcal{S}_1}\text{: } answer_1[j] \leftarrow \{(F(\mathbb{A}.v_j)_1 - F(q_1)) \times \mathcal{PRG}(seed_s[j]) + \mathcal{PRG}(seed_c[j])\} \bmod p$$
$$\boldsymbol{\mathcal{S}_2}\text{: } answer_2[j] \leftarrow ((F(\mathbb{A}.v_j)_2 - F(q_2)) \times \mathcal{PRG}(seed_s[j])) \bmod p$$

   where $F(\mathbb{A}.v_j)_z$ is the fingerprint of a value v (in additive share form) in the j^{th} row at $\mathcal{S}_z$. $\mathcal{S}_z$ sends $answer_z[]$ to the client. Note that $answer_z[]$ contains n integers regardless of the string length.

3) ***Client***: obtains the final answer of the search operator and executes: $vec[i] \leftarrow (answer_1[i] + answer_2[i]) \bmod p$. Also, the client executes $\mathcal{PRG}(seed)[i]_{i\in\{1,n\}}$. If $vec[i]$ matches $(\mathcal{PRG}(seed)[i]) \bmod p$, then query keyword exists at servers. Also, the client learns which row-ids contain the query keyword, and it will help to fetch the rows.[5]

5.1.2 *Example*

A client wants to know whether Tables 8, 9 contain "Jo" or not in the $\mathbb{A}(\texttt{Name})$ column. Client selects p=17 and r=2. Assume $\mathcal{PRG}(seed_c)$=$[4, 6, 1, 2]$ at $\mathcal{S}_1$, and $\mathcal{PRG}(seed_s)$=$[2, 9, 4, 5]$ at $\mathcal{S}_1, \mathcal{S}_2$. The single keyword search works as follows:

1) ***Client*:** creates shares and fingerprints of the query keyword Jo that is represented according to alphabet positions: $\langle 10, 15\rangle$. Additive shares of $\langle 10, 15\rangle$ are created: $\langle 5, 5\rangle$, $\langle 5, 10\rangle$. Finally, fingerprints are computed: $(5 \times 2 + 5 \times 2^2) \bmod 17 = 13$ and $(5 \times 2 + 10 \times 2^2) \bmod 17 = 16$. Fingerprint 13 is sent to $\mathcal{S}_1$ and fingerprint 16 is sent to $\mathcal{S}_2$.

2) ***Server*:** The first column of Table 10 (or Table 11) shows additive shares of the Name column at $\mathcal{S}_1$ (or at $\mathcal{S}_2$). The second column shows fingerprint computation over the Name column at $\mathcal{S}_1$ (or at $\mathcal{S}_2$). The third column shows the computation for searching Jo over fingerprints at $\mathcal{S}_1$ (or at $\mathcal{S}_2$). The fourth column shows the final result after using PRG. To client, $\mathcal{S}_1$ sends $\langle 14, 11, 6, 16\rangle$, and $\mathcal{S}_2$ sends $\langle 7, 15, 11, 16\rangle$.

3) ***Client*:** performs the following computation:
   $(14 + 7) \bmod 17 = 4$ $\quad$ $(11 + 15) \bmod 17 = 9$
   $(6 + 11) \bmod 17 = 0$ $\quad$ $(16 + 16) \bmod 17 = 15$
   The vector $\langle 4, 9, 0, 15\rangle$ is compared against $\mathcal{PRG}(seed_c)$=$[4, 6, 1, 2]$, and only the first position

5. While the client performs some computation on the received values, a majority of the computation is carried out on servers. Experiment 2 will show that the maximum processing time at the client is significantly less than 1s for 10M rows to know the qualified row-ids. Systems, e.g., Secrecy [52], which uses binary shares, also require the client to obtain n bits to know the row-ids. Systems, such as Jana and Conclave, transfer the job of the client to a trusted proxy for finding row-ids.

matches. This shows that the first row of the data contains the query keyword Jo. ■

NAME	Fingerprints of NAME	Search Computation	Final result
6,10	$6\times2+10\times2^2 \bmod 17=1$	$(1-13) \bmod 17=5$	$(5\times2+4) \bmod 17=14$
10,5	$10\times2+5\times2^2 \bmod 17=6$	$(6-13) \bmod 17=10$	$(10\times9+6) \bmod 17=11$
10,6	$10\times2+6\times2^2 \bmod 17=10$	$(10-13) \bmod 17=14$	$(14\times4+1) \bmod 17=6$
3,5	$3\times2+5\times2^2 \bmod 17=9$	$(9-13) \bmod 17=13$	$(13\times5+2) \bmod 17=16$

TABLE 10: Search computation at $\mathcal{S}_1$.

NAME	Fingerprints of NAME	Search Computation	Final result
4,5	$4\times2+5\times2^2 \bmod 17=11$	$(11-16) \bmod 17=12$	$(12\times2) \bmod 17=7$
3,10	$4\times2+10\times2^2 \bmod 17=12$	$(12-16) \bmod 17=13$	$(13\times9) \bmod 17=15$
2,9	$2\times2+9\times2^2 \bmod 17=6$	$(6-16) \bmod 17=7$	$(7\times4) \bmod 17=11$
10,10	$10\times2+10\times2^2 \bmod 17=9$	$(9-16) \bmod 17=10$	$(10\times5) \bmod 17=16$

TABLE 11: Search computation at $\mathcal{S}_2$.

5.1.3 Discussion

Now, we discuss correctness, information leakage, and cost related to the above algorithm.

Correctness. Recall that from the definition of fingerprint given in §1.1, the fingerprint function is additive homomorphic. Thus,

$$\begin{aligned}\Sigma_{i=1}^{i=2} answer_i[j] &= (\{F(A.v_j)-F(q_1)-F(q_2)\}\times\mathcal{PRG}(seed_s)[j]) \\ &\quad +\mathcal{PRG}(seed_c[j]) \bmod p \\ &= (\{F(A.v_j)-F(q)\}\times\mathcal{PRG}(seed_s))+(\mathcal{PRG}(seed_c)[j]) \bmod p\end{aligned}$$

Obviously, if the query keyword matches a value $A.v_j$, then the two fingerprints (*i.e.*, $F(A.v_j)$ and $F(q)$) will be identical, and the client receives only $(\mathcal{PRG}(seed_c)[j]) \bmod p$.

Information leakage discussion. We have already discussed in §4 that shares at-rest do not reveal frequency distribution and ordering of values. Now, let us discuss the security of query execution protocol. (*i*) Shares (or fingerprints) of the data at the servers and of query keywords are created randomly. Thus, a server by looking at the data and query keyword cannot learn which rows satisfy the query. (*ii*) Servers perform an identical operation on each row; this hides access-patterns. (*iii*) Each server sends n integers to the client, and this prevents volume leakage. (*iv*) Also, note that the client does not know $seed_s$. If values in two rows i and j do not match a query keyword, the client obtains two different random numbers (generated via $\mathcal{PRG}(seed_s)$), regardless of the values i and j are identical or not. In contrast, if the values match the keyword, the client always obtains zero. Therefore, the client learns only which rows match the query keyword and does not learn anything (*e.g.*, data distribution or ordering of values) about the secret-shared data.

Therefore, the single keyword search algorithm satisfies all security requirements, which are mentioned in §2.2.

Cost analysis. The computation cost at the server is $\mathcal{O}(n)$, while the client also adds n numbers received from each server. The communication cost between a server and a client depends on the size of n integers. §5.4 will provide a method to reduce the total communication cost from both servers to a client from $2n = 2\times n$ (n from each server) integers to only n integers.

5.1.4 Searching a Number

When searching a number, *e.g.*, age = 40, there is no need to create fingerprints. The client creates two additive shares of the number and sends them to servers. The servers execute the same computation as in Step 2 of the string matching operation, except for the fingerprint computation. In other words, servers directly subtract the received additive shares of a query from each additive share in the desired column of the data, without computing fingerprints. The client also performs the same computation on receiving n numbers from each server.

5.2 Conjunctive Search

In practical applications, queries involve multiple predicates over different columns. This section develops an approach for queries containing conjunctive predicates over multiple columns.

Consider a conjunctive search: `select * from Patient where name = 'Mo' and cost = 6`. A straightforward method to answer such queries is: to execute the single keyword search operator (§5.1) over each column at servers and sending multiple vectors (equal to the number of query predicates) containing n numbers in each, to the client, and then, the client locally finds the answer of the conjunctive search by finding one in each row over all the received vectors. While this trivial approach works, it incurs computational overhead and communication overhead of $\mathcal{O}(kn)$, where k is the number of conjunctive query predicates. To reduce such overhead to only n integers, we extend the single keyword search operator (§5.1) for $k > 1$ conjunctive predicates, as follows:

1) ***Client***: generates ***only one fingerprint regardless of the number of conjunctive conditions***. First, the client creates additive shares of each of the k predicates, depending on strings or numbers. Then, the client organizes k additive shares as a concatenated string and computes a single fingerprint over the string, as in STEP 1 of single keyword search §5.1.1. Fingerprints $F(q_z)_{z\in\{1,2\}}$ are sent to the server $\mathcal{S}_z$. $\mathcal{PRG}$ and $seed_c$ are also provided to $\mathcal{S}_1$.

2) ***Servers***: work on additive shares and execute the same operations as in §5.1.1 over each of the desired k columns of each row. Particularly, servers consider the k values of each j^{th} row as a string and compute a single fingerprint. Then, servers subtract the fingerprint received from the client and multiply $\mathcal{PRG}(seed_s)[j]$ to the output. Finally, $\mathcal{S}_1$ adds $\mathcal{PRG}(seed_c)[j]$ to j^{th} output. $\mathcal{S}_1, \mathcal{S}_2$ send outputs to client.

3) ***Client***: executes the same operation as in §5.1.1, *i.e.*, adds the elements of the received vector position-wise and compares against $(\mathcal{PRG}(seed_c)[i]) \bmod p$. If i^{th} values match, that means the k query predicates of the conjunctive search exist in the row i.

5.2.1 Example

A client wants to know whether Tables 8, 9 contain "name = Jo AND cost = 4" or not. Client selects $p = 17$, $r = 2$, and $\mathcal{PRG}(seed_c) = [4, 6, 1, 2]$. Assume $\mathcal{PRG}(seed_s) = [2, 9, 4, 5]$ at $\mathcal{S}_1, \mathcal{S}_2$. The conjunctive search operator works as follows:

1) ***Client*:** creates shares and fingerprints. $\langle$Jo, 4$\rangle$ is represented as: $\langle 10, 15, 4\rangle$. Additive shares are created as: $\langle 5, 5, 2\rangle$ $\langle 5, 10, 2\rangle$. Finally, fingerprints are created as: $(5\times2+5\times2^2+$

NAME	COST	Fingerprint computation	Search Computation	Final result
6,10	3	$6{\times}2+10{\times}2^2+3{\times}2^3 \bmod 17{=}8$	$(8{-}12) \bmod 17{=}13$	$(13{\times}2+4) \bmod 17{=}13$
10,5	2	$10{\times}2+5{\times}2^2+2{\times}2^3 \bmod 17{=}5$	$(5{-}12) \bmod 17{=}10$	$(10\times 9+6) \bmod 17{=}11$
10,6	4	$10{\times}2+6{\times}2^2+4{\times}2^3 \bmod 17{=}8$	$(8{-}12) \bmod 17{=}13$	$(13{\times}4{+}1) \bmod 17{=}2$
3,5	2	$3{\times}2+5{\times}2^2+2{\times}2^3 \bmod 17{=}8$	$(8{-}12) \bmod 17{=}13$	$(13{\times}5{+}2) \bmod 17{=}16$

TABLE 12: Search computation at $\mathcal{S}_1$ for conjunctive search.

NAME	COST	Fingerprint computation	Search Computation	Final result
4,5	1	$4{\times}2{+}5{\times}2^2{+}1{\times}2^3 \bmod 17{=}2$	$(2{-}15) \bmod 17{=}4$	$(4{\times}2) \bmod 17{=}8$
3,10	4	$3{\times}2{+}10{\times}2^2{+}4{\times}2^3 \bmod 17{=}10$	$(10{-}15) \bmod 17{=}12$	$(12{\times}9) \bmod 17{=}6$
2,9	4	$2\times 2{+}9\times 2^2{+}4{\times}2^3 \bmod 17{=}4$	$(4{-}15) \bmod 17{=}6$	$(6{\times}4) \bmod 17{=}7$
10,10	2	$10{\times}2{+}10{\times}2^2{+}2{\times}2^3 \bmod 17{=}8$	$(8{-}15) \bmod 17{=}10$	$(10{\times}5) \bmod 17{=}16$

TABLE 13: Search computation at $\mathcal{S}_2$ for conjunctive search.

$2{\times}2^3) \bmod 17{=}12$ and $(5{\times}2{+}10{\times}2^2{+}2{\times}2^3) \bmod 17{=}15$. Fingerprint 12 (15) is sent to $\mathcal{S}_1$ ($\mathcal{S}_2$).

2) ***Server*:** computation is shown in Table 12 and Table 13.

3) ***Client*:** performs the following computation on the received vectors $\langle 13, 11, 2, 16\rangle$ from $\mathcal{S}_1$ and $\langle 8, 6, 7, 16\rangle$ from $\mathcal{S}_2$:
$(13{+}8) \bmod 17{=}4, \quad (11{+}6) \bmod 17{=}0$
$(2{+}7) \bmod 17{=}9, \quad (16{+}16) \bmod 17{=}15$
Comparing the vector $\langle 4, 0, 9, 15\rangle$ against $\mathcal{PRG}(seed_c) = [4, 6, 1, 2]$ will show that the first row satisfies the conjunctive search. ■

5.2.2 *Discussion*

Information leakage discussion. Let us discuss information leakage from query executions. First, servers do not learn query predicates by just observing fingerprints received from the client, due to working on additive shares. Second, on each of the k columns of each row, a server performs an identical operation that hides access-patterns. Also, the output at each server will be different for each row, regardless k query predicate matches or not in multiple rows; thus, the output at each server does not reveal anything about the final result. Third, each server sends n numbers to the client, and it prevents volume leakage. Finally, since the client is not aware of $seed_s$, the client only learns which rows satisfy the query, nothing else.

Cost analysis. The computation cost at the server is $\mathcal{O}(n)$. Regardless of the number of columns involved in a conjunctive query, the client works on n numbers received from each server, and the communication cost depends on the size of n integers, (as in the case of the single keyword search algorithm).

5.3 Disjunctive Search

A disjunctive search (`select * from table where name = 'Jo' or cost = 4`) finds all those rows that satisfy multiple query predicates connected using 'or' over different columns.

High-level idea. *This approach works over multiplicative shares.* Let a, b, c be three values in three different columns of a table. If query predicates are either a, b, or c, then subtraction of the query predicate will result in $a = 0$, $b = 0$, or $c = 0$, and then, $a \times b \times c = 0$. We do exactly the same over multiplicative shares — servers subtract the query keyword and multiply the answer.

Step-wise details. We provide detailed steps of disjunctive search:

1) ***Client*:** creates multiplicative shares of k query predicates using polynomials of degree one and sends them to servers. For $k = 2$ (and $k \geq 3$) predicates, three (and four) multiplicative shares are created and sent to $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ (and all four servers).

2) ***Servers*:** subtract the k query predicates from each value of the desired column, which is also in multiplicative share forms, and then, multiply the output of any three columns (*i.e.*, $\lceil k/3 \rceil$). Note that $k = 2$ is a special case, and here, the output of two columns are multiplied at $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$. In other words, if there are $3k$ columns, then servers, after subtraction, execute multiplication over groups of three columns in each. Such multiplication will result in a polynomial of degree three. Finally, the server multiplies $\mathcal{PRG}(seed_s)[i]$ and adds $\mathcal{PRG}(seed_c)[i]$. Finally, servers send $\lceil k/3 \rceil$ vectors to the client.

3) ***Client*:** performs Lagrange interpolation on each $\lceil k/3 \rceil$ vector and matches the i^{th} position of each vector against $\mathcal{PRG}(seed_c)[i]$. A match shows that the i^{th} row satisfies the disjunctive query. (Since SEASEARCH uses at most four servers, the client can interpolate the shares of four servers to recover the answer.)

A	B
1	5
6	4

TABLE 14: An input cleartext table.

A	B
2	8
8	8

TABLE 15: Share Table$_1$.

A	B
3	11
10	12

TABLE 16: Share Table$_2$.

A	B
4	14
12	16

TABLE 17: Share Table$_3$.

5.3.1 *Example.*

Table 14 shows a cleartext table, whose multiplicative secret-shares are shown in Tables 15, 16, and 17. Consider a prime number $p = 17$. For creating multiplicative shares of 1, DBO uses $f(x) = (1x + s) \bmod p$, where $s = 1$ is the secret value, and obtains shares: $f(1) = 2$, $f(2) = 3$ and $f(3) = 4$. For creating multiplicative shares of 5, DBO uses $f(x) = (2x{+}5) \bmod p$ and obtains shares: $f(1) = 8$, $f(2) = 11$ and $f(3) = 14$. For creating

A	B	Search Computation	Final result
2	8	$((2-4) \bmod 17 \times (8-7) \bmod 17) \bmod 17 = 15$	$(15+1) \bmod 17 = 16$
8	8	$((8-4) \bmod 17 \times (8-7) \bmod 17) \bmod 17 = 4$	$(4+2) \bmod 17 = 6$

TABLE 18: Search computation at $\mathcal{S}_1$ for disjunctive search.

A	B	Search Computation	Final result
3	11	$((3-7) \bmod 17 \times (11-8) \bmod 17) \bmod 17 = 5$	$(5+1) \bmod 17 = 6$
10	12	$((10-7) \bmod 17 \times (12-8) \bmod 17) \bmod 17 = 12$	$(12+2) \bmod 17 = 14$

TABLE 19: Search computation at $\mathcal{S}_2$ for disjunctive search.

A	B	Search Computation	Final result
4	14	$((4-10) \bmod 17 \times (14-9) \bmod 17) \bmod 17 = 4$	$(4+1) \bmod 17 = 5$
12	16	$((12-10) \bmod 17 \times (16-9) \bmod 17) \bmod 17 = 14$	$(14+2) \bmod 17 = 16$

TABLE 20: Search computation at S_3 for disjunctive search.

multiplicative shares of 6, DBO uses $f(x) = (3x+8) \bmod p$ and obtains shares: $f(1) = 8$, $f(2) = 10$ and $f(3) = 12$. For creating multiplicative shares of 4, DBO uses $f(x) = (4x+4) \bmod p$ and obtains shares: $f(1) = 8$, $f(2) = 12$ and $f(3) = 16$.

Consider a disjunctive search: `select * from Test where A = 1 or B = 6`. A client wants to know whether Table 15, 16, and 17 contain "A = 1 OR B = 6" or not. The client selects $p = 17$ and $\mathcal{PRG}(seed_c) = [1, 2]$. The disjunctive search operator works as follows:

1) ***Client***: creates multiplicative shares of 1 and 6 using polynomials of degree one. Shares of 1 are obtained as: $4, 7, 10$ using $f(x) = (3x+1) \bmod p$, for $x = 1, 2, 3$. Shares of 6 are obtained as: $7, 8, 9$ using $f(x) = (1x+6) \bmod p$, for $x = 1, 2, 3$. Shares $\langle 4, 7\rangle$ are sent to $\mathcal{S}_1$; $\langle 7, 8\rangle$ are sent to $\mathcal{S}_2$; and $\langle 10, 9\rangle$ are sent to $\mathcal{S}_3$.
2) ***Servers***: performs computation over data and query predicates both in multiplicative share form. Tables 18, 19, and 20 show computation at each server. For simplicity, we do not show $\mathcal{PRG}(seed_s)$.
3) ***Client***: receives vectors $\langle 16, 6\rangle$ from $\mathcal{S}_1$, $\langle 6, 14\rangle$ from $\mathcal{S}_2$, and $\langle 5, 16\rangle$ from $\mathcal{S}_3$. On such values, the client performs Lagrange interpolation as follows:

$$f(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} \times y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} \times y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} \times y_3 \quad (1)$$

For $y_1 = 16$, $y_2 = 6$ and $y_3 = 5$, $f(x)$ evaluates to:

$$f(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} \times 16 + \frac{(x-1)(x-3)}{(2-1)(2-3)} \times 6 + \frac{(x-1)(x-2)}{(3-1)(3-2)} \times 5 = 35 \bmod 17 = 1 \quad (2)$$

For $y_1 = 6$, $y_2 = 14$ and $y_3 = 16$, $f(x)$ evaluates to:

$$f(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} \times 6 + \frac{(x-1)(x-3)}{(2-1)(2-3)} \times 14 + \frac{(x-1)(x-2)}{(3-1)(3-2)} \times 16 = -8 \bmod 17 = 9 \quad (3)$$

Comparing $\langle 1, 9\rangle$ against $\mathcal{PRG}(seed_c)$ = [1,2] shows that the first row satisfies the disjunctive search.

5.3.2 *Discussion*

Information leakage discussion. A server cannot learn data and query predicates by observing them, since both are in multiplicative share form. Also, the shares of data and shares of query predicate are not identical, due to using different polynomials (of the same degree). A server cannot learn whether any value resulted in zero after subtraction, since the output of subtraction is also in multiplicative share form. Since servers perform an identical operation on each row, access-patterns are not revealed. Servers return n numbers, and the number of rows satisfying a query is also not revealed.

Cost analysis. Communication cost between a server and a client is $\lceil kn/3 \rceil$), where k is the number of disjunctive query predicates. Computation cost at a server is at most kn, and at a client is $\lceil kn/3 \rceil$).

5.4 Optimizing Communication Cost

In the search algorithms developed in the previous subsections, a client receives a vector of n numbers from both servers $\mathcal{S}_1, \mathcal{S}_2$ in the case of single keyword and conjunctive search, while may receive $\lceil k/3 \rceil$ vectors (each with n numbers) from $\mathcal{S}_1, \ldots, \mathcal{S}_4$, ($k$ is the number of query predicates in a disjunctive search). In practical situations, a client may be geographically far away from servers and/or has a limited network connection speed and/or hold a weaker machine. In such a case, the following can happen: (*i*) the processing time at the client will increase, and/or (*i*) transferring the data from the servers to the client may increase the overall query processing time. To reduce the computation time at the client and communication between a server and a client, we provide a method that requires an additional untrusted server, called ***combiner server***, denoted by $\mathcal{S}_c$; see Figure 2.

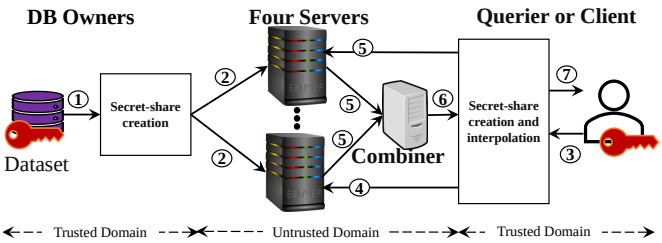


Fig. 2: The model with combiner.

Before presenting the method, let us discuss assumptions behind $\mathcal{S}_c$. Firstly, $\mathcal{S}_c$ is *not trusted* likewise servers $\mathcal{S}_1, \ldots, \mathcal{S}_4$. $\mathcal{S}_c$ only knows the modulus used in the fingerprint. $\mathcal{S}_c$ receives shares from servers and computes modular addition or Lagrange

interpolation. While $\mathcal{S}_c$ knows both servers, we assume that S_c never sends the data received from one server to another. Also, servers will not send the additive shares to $\mathcal{S}_c$. Such requirements are necessary to prevent $\mathcal{S}_c$ or servers to reconstruct the secret, i.e., the original data in cleartext. Likewise, other servers $\mathcal{S}_1, \ldots, \mathcal{S}_4$, $\mathcal{S}_c$ also wishes to learn about the original data, based on the received shares and the computation it does. Thus, we need to prevent access-patterns and volume at $\mathcal{S}_c$ also and ***must need*** to satisfy our Security Definition 1. The role of the combiner has been also considered in the previous work, for example, [16], [25], [45], [72], [74].

Method. This method is quite straightforward. Now, servers send the output of the computation to $\mathcal{S}_c$ instead of sending it to the client. Importantly, now, servers ***must use*** $\mathcal{PRG}(seed_c)$ in STEP 2 of the respective algorithms.

In the case of *single keyword search or conjunctive search*, $\mathcal{S}_c$ performs modular addition under p and sends a single vector to the client that matches the i^{th} received vector's value against $\mathcal{PRG}(seed_c)[i]$, $1 \leq i \leq n$. The matching i^{th} index shows that the i^{th} row at the server satisfies the query. Note that now $\mathcal{S}_c$ sends only n numbers to the client, and the client *does not* need to perform modular addition.

In the case of *disjunctive search*, $\mathcal{S}_c$ performs Lagrange interpolation over $\lceil k/3 \rceil$ vectors and sends interpolated vectors to client that compares the i^{th} value of the vector against $\mathcal{PRG}(seed)[i]_{1 \leq i \leq n}$, and the matching i^{th} index shows the i^{th} row satisfies the query.

Security discussion. $\mathcal{S}_c$ must never learn the final answer after performing modular addition or Lagrange interpolation. Since $\mathcal{S}_c$ does not know $\mathcal{PRG}(seed_c)$ added by servers, $\mathcal{S}_c$ finds all n numbers to be random. Thus, $\mathcal{S}_c$ cannot learn which rows satisfy the query. Also, $\mathcal{S}_c$ performs an identical operation on each share received from servers and always sends a vector of n numbers to the client. Thus, access-patterns and volume are hidden from $\mathcal{S}_c$.

Reason for adding $\mathcal{PRG}(seed_c)$ at servers. Until now, one can check the necessity of $\mathcal{PRG}(seed_c)$: if the server does not add such random numbers, then $\mathcal{S}_c$ will learn which rows and how many rows satisfy the query.

5.5 Multi-Keyword Search

A multi-keyword search query finds different keywords in a column, simultaneously. For example, `select * from table where name = 'Jo' or name = 'Mo'` will fetch all rows containing Jo or Mo. Such a query can be answered by executing the simple search operator multiple times, equal to the number of query keywords. However, this straightforward method incurs overheads in terms of both computation and communication. To reduce the overhead, we develop an algorithm for multi-keyword search that takes only *one communication round*.

High-level idea. In searching multiple keywords, now the client sends additive shares of the keywords, and the server computes $x > 1$ fingerprints over the additive shares of the keyword. Also, servers compute fingerprints over additive shares in the desired column and send the output vector to $\mathcal{S}_c$ that adds the output vectors position-wise and provides a single vector to the client. Finally, the client compares the fingerprints of the keyword, which were also received from the server, over the vector, which is received from $\mathcal{S}_c$. This way, we search for multiple keywords.

Sending the vector (containing the fingerprints of the column) from $\mathcal{S}_c$ to the client will reveal data distribution to the client, and to hide this, the servers compute $x > 1$ fingerprints on the client queries by adding x random numbers.

The method works as follows:

1) *Client*: creates additive shares of the $k > 1$ keywords and sends them to $\mathcal{S}_1$ and $\mathcal{S}_2$. Note that here, the client does not compute fingerprints over additive shares of the query keyword, unlike the single keyword search algorithm §5.1.

2) *Servers*: perform three tasks: (*i*) select $x \leq n$ random numbers in $\mathbb{F}_p$, (*ii*) compute x fingerprints for each $k > 1$ query keyword by adding each of the x random numbers to the fingerprints, and send all such fingerprints $queryFP[]$ to the client, *note that here the servers compute fingerprints, and the fingerprint parameter used by servers is not known to the client*, and (*iii*) compute fingerprints over additive shares of the desired column, as follows:
   $\mathcal{S}_1$: $\forall i \in block_j$: $answer_1[i] \leftarrow F(\mathbb{A}.v_i) + j^{th}$ random number
   $\mathcal{S}_2$: $\forall i \in block_j$: $answer_2[i] \leftarrow F(\mathbb{A}.v_i) + j^{th}$ random number
   The servers partition the data into x blocks, compute fingerprints over the additive shares of the desired columns, and add the j^{th} random number to the output of fingerprint for each row in the j^{th} block. This computation produces a vector $answer_{z \in \{1,2\}}[]$ that is sent to the client.

3) *Combiner*: executes $vec[] \leftarrow answer_1[i] + answer_2[i]$ and sends the resulting vector $vec[]$ to the client.

4) *Client*: compares the received vector $vec[]$ against all the fingerprints $queryFP[]$. The client compares all j^{th} fingerprints of each $k > 1$ keywords against all the rows of the j^{th} block of $vec[]$. If a fingerprint matches, it means the keyword exists.

Method for multiple numeral search. The above method can be extended trivially for searching multiple numbers.

Information leakage. Likewise, the arguments for the servers in the single keyword search method in §5.1.1, we can derive arguments for the server in the multi-keyword search method. Specifically, the servers do not learn frequency distribution, ordering, and access-patterns.

Now, we discuss the case of $\mathcal{S}_c$ and the client. Note that $\mathcal{S}_c$ holds fingerprints of each value of the column, added with random numbers. Based on fingerprints, $\mathcal{S}_c$ can never know data, as it needs to guess the parameter r used to compute fingerprints and the random numbers x_i added to fingerprint ($F(\mathbb{A}.v_i)$). Further, since the entire data is divided into multiple blocks, the output (*i.e.*, $\sum_{z=1}^{z=2} answer_z[i]$) across the blocks for a keyword will be non-identical, due to using different random numbers for each block. Thus, $\mathcal{S}_c$ cannot learn the data distribution. $\mathcal{S}_c$ performs identical operations on each value of the received vector from the servers and returns the same number of values to the client. Thus, access-patterns and volume are not revealed to $\mathcal{S}_c$. To make the technique more secure at $\mathcal{S}_c$, the servers can add $\mathcal{PRG}(seed_c)$ before sending the vector to $\mathcal{S}_c$. Now, to learn anything from fingerprints, $\mathcal{S}_c$ needs to know multiple PRGs.

Now, let us discuss leakages at the client. The client learns only the rows that match the keywords, nothing else. The client does not learn anything from $vec[]$ due to not knowing x random

numbers. Thus, to the client, each number in $vec[]$ will look different, preventing the client from learning the data distribution.

Cost analysis. The communication cost between the client and $\mathcal{S}_c$ is $\mathcal{O}(n)$, and between a server and $\mathcal{S}_c$ is also $\mathcal{O}(n)$. The communication cost can grow if the client wishes to search for many keywords and the server creates many blocks over the data. The computation cost at each entity is $\mathcal{O}(n)$, as executing computation over n values of a column.

5.6 Range Search

A range search outputs a list of values (and row ids) that satisfy a given range query. We develop two methods to answer a range query. Both methods use additive shares. The first method is communication efficient, but may reveal the order of values to the client. The second method is completely secure and does not reveal any information to the client; however, incurs the communication cost ($\mathcal{O}(x\ell)$, where ℓ is the length of a range query and $x>1$).

We describe a range query over integers, while the same idea applies to non-integers also. Let $[\mu_1, \mu_2]$ be the endpoints of a range. A value v belongs to the range, iff $\mu_1 \leq v \leq \mu_2$. Our primary idea of both approaches is to generate a unique value for each number in the range $[\mu_1, \mu_2]$ by executing a function.

5.6.1 *Communication-Efficient Range Query*

High-level idea. In this method, we check that $v \in [\mu_1, \mu_2]$, iff $f(\mu_1) \leq f(v) \leq f(\mu_2)$, where f denotes an order-preserving function. For example, 3,4,5 belong to a range $[3, 5]$. The output of a function $f(x)=x+\mu_1^{\mu_2}$ can also detect whether $3, 4, 5 \in [3, 5]$ or not, since $f(3), f(4), f(5) \in [f(3), f(5)]$, *i.e.*, $246, 247, 248 \in [246, 248]$, and no other number can belong to $[f(3), f(5)]$.

The servers compute $x > 1$ different functions on the range end-points and provide the output to the client. The servers also compute the x different functions on x different parts of the data (which is in additive share form) and send the output to $\mathcal{S}_c$ that adds the received data from servers row-wise. Finally, $\mathcal{S}_c$ sends the vector to the client, which finds which row ids belong to the range query. The method works as follows:

1) *Client*: creates additive shares of the range queries' end-points, say μ_1, μ_2, and sends them to the servers.

2) *Servers*: do the following:
(*i*) find x different functions $(f_1, \ldots, f_x)$ and execute them on the range end-points μ_1, μ_2: $op_{zi1} \leftarrow f_i(\mu_1)$, $op_{zi2} \leftarrow f_i(\mu_2)$, where op_{zi1} and op_{zi2} are the output of the function f_i on the range endpoints, and $z \in \{1, 2\}$. The outputs of the x functions, *i.e.*, $1 \leq i \leq x : op_{zi1}, op_{zi2}$, are provided to the client.
(*ii*) work over additive shares of each value of the column. For simplicity, we assume that secret-shared data will be partitioned into x blocks. For each block, servers select a function.
Servers compute: $answer_z[j] \leftarrow f_i(\mathbb{A}.v_j) + \mathcal{PRG}(seed_c)$, *i.e.*, the function f_i, on each row of the i^{th} block, is computed, and a vector $answer_z[]$ is sent to $\mathcal{S}_c$.

3) *Combiner*: adds $answer_1[i]$ and $answer_2[i]$ and obtains a vector, $vec[]$ that is sent to the client.

4) *Client*: subtracts $\mathcal{PRG}(seed_c)$ from each value of $vec[]$ and then partitions $vec[]$ into x blocks. Over an i^{th} block, the client finds all those numbers that belong to the range $[op_{i1}, op_{i2}]$, where $op_{i1} = \sum_{z=1}^{z=2} op_{zi1}$ and $op_{i2} = \sum_{z=1}^{z=2} op_{zi2}$.

Correctness. It is clear that the function pairs $(op_{i1}, op_{i2}) = \{f_i(\mu_1) \odot_i f_i(\mu_2)\}$ are all linear functions, so they are order-preserving. One can easily check that if $v \in [\mu_1, \mu_2]$, then $f_i(v_j) \in [f_i(\mu_1), f_i(\mu_2)]$.

Cost analysis. Communication cost: (*i*) from the server to the client is $\mathcal{O}(2x)$, (x is number of blocks), by sending $2x$ range end-points, (*ii*) from $\mathcal{S}_c$ to the client is $\mathcal{O}(n)$, (*iii*) between a server and $\mathcal{S}_c$ is also $\mathcal{O}(n)$, by sending n values. The computation cost at each entity is $\mathcal{O}(n)$.

Information leakage. Servers cannot learn anything, due to executing an identical operation on each row in the form of additive shares. $\mathcal{S}_c$ does not learn anything, since servers add $\mathcal{PRG}(seed_c)$. The client can learn the difference between $f(x_1) - f(x_2)$. If the client knows some values in the data, then they may deduce secret values based on $f(x_1) - f(x_2)$.

5.6.2 *Fully-Secure Range Query*

To prevent the client from learning $f(x_1) - f(x_2)$, this method requires the DBO to send additional data to servers. Particularly, DBO executes a PRG function on each value of the domain of a column and also outsources such values, denoted by $\mathcal{PRG}_{dbo}(v)$, in additive share form to the servers. Furthermore, the DBO represents all column values using $v + \mathcal{PRG}_{dbo}(v)$ and creates an additional type of shares. Thus, now, DBO outsources all columns of a table in three forms: additive shares, multiplicative shares, and additive shares of $v + \mathcal{PRG}_{dbo}(v)$. To execute completely secure range queries, we use additive shares of $v + \mathcal{PRG}_{dbo}(v)$.

Three differences between this method and the previous method are: (*i*) The servers compute x functions on *each number* in a given range $[\mu_1, \mu_2]$ in additive share form, sent by the client, as follows: $op_{zi}[j] \leftarrow f_i(a_j + \mathcal{PRG}(a_j))$, and sends the vector $op_{zi}[]$ containing all the possible outputs of a function f_i to the client; (*ii*) the servers work on the additive shares of $v + \mathcal{PRG}_{dbo}(v)$ and perform the same operation, as in the previous method; and (*iii*) on receiving the vector from $\mathcal{S}_c$, the client finds all those numbers that belong to $op_i[j]$. The method works as follows:

1) *DBO*: creates additive shares, $\mathcal{PRG}_{dbo}(v)$, for each domain value of the attribute. Such shares are outsourced separately in a different table. Furthermore, DBO creates additive shares of $v_i + \mathcal{PRG}_{dbo}(v_i)$, where v_i is a value in the i^{th} row of the column.

2) *Client*: first finds additive shares of $\mathcal{PRG}_{dbo}(v)$ each value $v \in [\mu_1, \mu_2]$. Then, the client creates additive shares of each value $v + \mathcal{PRG}_{dbo}(v)$ in range queries and sends them to the servers.

3) *Servers*: do the following:
(*i*) Find x different functions $(f_1, \ldots, f_x)$ and execute them on each additive share received from the client. This step produces a vector: $op_{zi}[]$ at $\mathcal{S}_z$. Such a vector is given to the client.
(*ii*) Work over additive shares of a number. For simplicity, we assume that secret-shared data will be partitioned into x blocks. For each block i, servers select a function f_i and

compute: $answer_z[j] \leftarrow f_i(\mathbb{A}.v_j + \mathcal{PRG}_{dbo}(v_j))$, *i.e.*, the function f_i, on each row of the i^{th} block, is computed, and this vector $answer_z[]$ is sent to $\mathcal{S}_c$.

4) *Combiner*: adds $answer_1[i]$ and $answer_2[i]$ and obtains a vector, $vec[]$ that is sent to the client.

5) *Client*: partitions $vec[]$ into x blocks. Over an i^{th} block, the client finds all those numbers that belong to $\{op_{i1}, op_{i2}\}$.

Cost analysis. Communication cost from the client to the server is $\mathcal{O}(\ell)$, where ℓ is the length of a range query. Communication cost from a server to the client is $\mathcal{O}(x\ell)$, where x is the number of blocks the servers create. Communication cost from $\mathcal{S}_c$ to the server is $\mathcal{O}(n)$, and the same communication cost exists between servers and $\mathcal{S}_c$. Computation cost both at servers and $\mathcal{S}_c$ is $\mathcal{O}(n)$.

Information leakage. In this method, servers do not learn anything, likewise the method of §5.6.1. Also, the combiner does not learn anything, due to using $\mathcal{PRG}_{dbo}(v_j)$ and x blocks. Furthermore, the client cannot learn anything within a block, due to using $\mathcal{PRG}_{dbo}(v_j)$, as PRG will eliminate any relationship between two numbers.

5.7 Conjunctive or Multi-dimensional Range Search

This subsection presents two approaches for answering multi-dimensional range queries (`select * from employees where age between 10 and 20 and salary between 100 and 200`). These proposed approaches follow the idea of a one-dimensional range query given in §5.6. Both approaches work on additive shares of a number. The first approach generates a unique value for the combinations of different *range end-points*. The second approach works on additive shares of $v + \mathcal{PRG}_{dbp}(v)$ and generates a unique number for *each value belonging to a range*. Below, we explain the idea using a two-dimensional range condition.

The first approach. Let $(\mu_1^1 \leq A_1 \leq \mu_2^1) \wedge (\mu_1^2 \leq A_2 \leq \mu_2^2)$ be a two-dimensional range query, where we assume that $\mu^*_{*\in\{1,2\}}$ have the same order of magnitude and the superscript refers to column and subscript refers to range. Our aim is to construct a function $f(A_1, A_2, \mu_1^1, \mu_2^1, \mu_1^2, \mu_2^2)$ that produce unique outputs for different values of A_1 and A_2. We use a simple way: $f(A_1, A_2, \mu_1^1, \mu_2^1, \mu_1^2, \mu_2^2) = (A_1 \odot (\rho_{11}\mu_1^1 + \rho_{12}\mu_2^1) + (A_2 \odot (\rho_{21}\mu_2^2 + \rho_{22}\mu_2^2)) \times r) \bmod p$ where $\rho_{*,*}$ are random numbers, and r is another random number that makes $(\rho_{21}\mu_1^2 + \rho_{22}\mu_2^2) \times r$ much bigger than $\rho_{11}\mu_1^1 + \rho_{12}\mu_2^1$. This feature makes the function f outputting different results according to different inputs A_1, A_2. In addition, if the range $[\mu_1^1, \mu_2^1]$ and $[\mu_1^2, \mu_2^2]$ are not in the same order of magnitude, we can adjust the value of r to satisfy our requirement.

For example, for a range query for $2 \leq A_1 \leq 3$ and $5 \leq A_2 \leq 7$, we construct a function as follows: $F(A_1, A_2, 2, 3, 5, 7) = (A_1 - (9\times 2 + 4 \times 3) + (A_2 - (1 \times 5 + 8 \times 7)) \times 10) \bmod p$. Assume $p = 13$, we immediately know that for all the value pairs $(2,5), (2,6), (2,7), (3,5), (3,6), (3,7)$ in the conjunctive range, the above function produces different values $\{10, 7, 4, 11, 8, 5\}$.

Now, the client and servers follow the same protocol on the multi-dimensional range queries' end-point as they did in 5.6.1.

The second approach: is the extension of the fully-secure range search §5.6.2. Here, servers compute x different functions $f_1 \ldots, f_x$ on each pair of values (*i.e.*, f_k over $i + \mathcal{PRG}_{dbo}(i), j + \mathcal{PRG}_{dbo}(j)$, where $\forall i, j \in [\mu_1^1 \leq A_1 \leq \mu_2^1 \wedge \mu_1^2 \leq A_2 \leq \mu_2^2]$), in additive share form and sends the output to the client. Also, servers compute functions on the given columns and provide the vector $\mathcal{S}_c$, which adds the vector position-wise. The client on receiving the vector from $\mathcal{S}_c$ finds elements that match against the list provided by the servers for each value of a range query.

6 Fetch Operator

A fetch operator retrieves the desired rows containing a keyword. To do so, first, the client needs to know the row id using the search operators. Afterward, the client needs to fetch the row(s) obliviously. A straightforward way for oblivious fetch is private information retrieval (PIR). It is worth noting that, in our setting, all servers store data in shares form, not cleartext. Thus, any PIR schemes should be modified accordingly, in order to fetch shares obliviously. On top of that, for practical implementations, we also need to use a limited number of servers (unlike existing work [39]). To achieve these goals, we develop two methods: *multiplicative sharing-based method* (§6.1) and *PRG-based method* (§6.2) that uses additive shares. The multiplicative sharing-based method simplifies the model of [39] and provides a practical scheme. The additive sharing method extends DPF [38], which was designed for cleartext processing. Both these methods utilize four servers and are compared in §6.3.

6.1 Multiplicative Sharing-based Method

						Row id
0	0	$\cdots$	0	$\cdots$	0	1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	0	$\cdots$	**1**	$\cdots$	0	***i***
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	0	$\cdots$	0	$\cdots$	0	$\sqrt{n}$
1	2	$\cdots$	***j***	$\cdots$	$\sqrt{n}$	Column ids

High-level idea. Given a cleartext vector, vec, of size n, containing all zeros, except a single one at the position that the client wishes to fetch, we organize vec into a matrix of r rows and c columns. For the purpose of simplicity, here, we assume that $r = c = \sqrt{n}$.[6] Thus, only one of the cells (i, j) (*i.e.*, row i and column j) of the matrix contains one; otherwise, zero, as shown in the matrix.

Now, we create two vectors: $r_1 = \langle 0, \ldots, 0, 1_i, 0, \ldots, \sqrt{n}\rangle$ and $r_2 = \langle 0, \ldots, 0, 1_j, 0, \ldots, \sqrt{n}\rangle$. If we position-wise multiply each row of the matrix by r_1, then we will obtain the i^{th} row of the matrix after adding values of each column of the matrix. Now, we can multiply the resultant row by r_2 to get the desired value/row.

SeaSearch implements exactly the same. Particularly, the client sends r_1 and r_2 vectors in multiplicative share form, and

6. The reason for selecting a grid/matrix of $\sqrt{n} \times \sqrt{n}$ will be clear in the communication cost analysis. In case, when $\sqrt{n}$ results in a non-integer number, we find two numbers, say x and y, such that $x \times y = n$, and x and y are equal or close to each other such that the difference between x and y is less than the difference between any two factors, say x' and y' of n so that $n = x' \times y'$.

each of the four servers implements exactly the same idea over multiplicative shares.

Details of the methods are given below:

1) ***Client*:** creates two *row vectors* r_1 and r_2, each of size $\sqrt{n}$ and filled with zeros. Suppose, the client wants to fetch a row that is mapped to the $(i,j)^{th}$ cell of the matrix of size $\sqrt{n} \times \sqrt{n}$, then the i^{th} value of r_1 and the j^{th} value of r_2 contain 1. The client creates multiplicative shares (or SSS) of r_1 and r_2 and sends them to four servers.

2) ***Servers*:** organizes the data in a form of $\sqrt{n}\times\sqrt{n}$ matrix and multiply the k^{th} values of r_1 with each tuples in the k^{th} row of the matrix. After that, servers adds all attribute across all rows of each column of the matrix, resulting in a single row containing $\sqrt{n}$ tuples. Finally, to the single row, servers position-wise multiply r_2 vectors and add the output of each attribute across $\sqrt{n}$ tuples, if the client wants to fetch only $(i,j)^{th}$ row.[7] Servers send the final output to the client.

3) ***Client*:** receives shares of the desired row from the servers and performs Lagrange interpolation to get the real values.

Information leakage. Based on the row vectors, this method can never reveal to servers which row they return, since the row vectors are in share form. Since servers perform identical computations on each row, this also prevents access-patterns. Furthermore, servers return either one row (in case of the client wants only one row) or $\sqrt{n}$ rows (otherwise), which prevents volume leakage.

Cost analysis. The computation cost at the server is $\mathcal{O}(n)$, while at the client is $\mathcal{O}(x)$, where $x \in \{1, \sqrt{n}\}$. Each row vector contains $\sqrt{n}$ numbers, which enable the client to fetch $x \in \{1, \sqrt{n}\}$ rows. Note that if all the rows containing a query keyword exist in either the same row or the same column of the matrix, we can fetch all of them in a single communication round. Thus, to fetch $x \in \{1, \sqrt{n}\}$ rows, the communication cost is $\mathcal{O}(\sqrt{n})$. Since the minimal communication cost can be achieved by organizing n tuples in a matrix of the minimum size that can be achieved by a $\sqrt{n}\times\sqrt{n}$ matrix. Thus, we create a matrix of $\sqrt{n}\times\sqrt{n}$.

6.2 Additive Share-based Method

[27] provided a trivial private information retrieval (PIR) scheme that allows a client to obliviously obtain one bit from two servers. This method can be extended to fetch an $i^{th}_{i\in\{1,n\}}$ additive share using four servers, as follows:

Bit vector-based method. Assume that there are two numbers $a, b\in\{0,1\}$ such that $a-b$=1 or 0 (depending on the value $a-b$). When we multiply $(a-b)$ by a value $X = x_1+x_2$, the product, say z, will remain X or zero. Meanwhile, the expansion $(x_1+x_2)(a-b) = ax_1+ax_2-bx_1-bx_2$ can be split into four parts, and each part can be executed over one of the four servers *locally*. Note that a server having partial information cannot learn the final result z, while a client with ax_1, ax_2, bx_1, bx_2 can know z. This method can be used to fetch a k^{th} row by creating two *row vectors* r_1 and r_2 of size n, such that $r_1[i] - r_2[i] = 0, \forall i \in \{1,n\}\backslash\{k\}$ and $r_1[k] - r_2[k] = 1$.

Example. Suppose a dataset contains $\{4, 2, 6\}$, and a client wishes to fetch the third element. Assume that the dataset is stored in additive shares at four servers. The client can create row vectors: $r_1 = [0, 1, 1]$ and $r_2 = [0, 1, 0]$, and send each vector to two servers. Each server will perform position-wise multiplication and then add all values before returning a single value to client, which learns the output. ■

Problem of bit vector-based method. This method incurs the high communication cost of n-bits to fetch a single row. To reduce the communication cost, below, we propose a new method:

6.2.1 High-level Idea

Our objective is to compress the row vectors from n-bits to $\log n$-bits at the client; while, at the server, to decompress such vectors to size n, each. To do so, we identify Distributed Point Function (DPF) [38] as a natural fit for our oblivious fetch scheme. DPF was designed to fetch a single value from cleartext data, without revealing the value. We extend DPF to work over additive shares. To do so, row vectors r_1 and r_2 can be recognized as the additive shares of a point function $f(x)$, to fetch k^{th} row, where $f(x) = 1$ if $x = k$; otherwise, $f(x) = 0$.

To compress shares of r_1 and r_2, we do the following at a client:

Idea of compression at the client. We provide the idea to compress n-bits to $\sqrt{n}$ bits, which can be recursively compressed into $\log n$ bits. First, organize n bits in a matrix, likewise the method of §6.2. Thus, only one of the cells (i,j) (*i.e.*, row i and column j) of the matrix contains one; otherwise, zero. Then, we can form two row vectors r_1 and r_2. Obviously, except the i^{th} row and j^{th} column of the matrix, the rest parts of r_1, r_2 contain only zeros.

To hide the actual values (*i.e.*, 0 or 1), we use PRG with $\sqrt{n} - 1$ different seeds, denoted by $sd_1, sd_2, \ldots, sd_{i-1}, sd_{i+1}, \ldots, sd_{\sqrt{n}}$ (one seed generates a random number for a tuple of the data). Note that $\mathcal{PRG}(sd_q) - \mathcal{PRG}(sd_q) = 0$, $\forall q \in \{1, 2, \ldots, i-1, i+1, \ldots, \sqrt{n}\}$.

Now, to construct the i^{th} row of the matrix, we need two more vectors: cr_α, cr_β, each of size $\sqrt{n}$, such that

$$\mathcal{PRG}(sd_\delta)[k] + cr_\alpha[k] - (\mathcal{PRG}(sd_\theta)[k]+cr_\beta[k]) = 1, \text{ for } k=j$$
$$\text{otherwise, for } k \neq j = 0.$$

We call cr_α, cr_β as ***compressed row vectors***. In the following, for the purpose of simplicity, we denote (compressed) row vectors as r_α and r_β. Note that by only looking at the value of r_α or r_β, no one can find which value will result in one or zero.

Finally, the client forms two seed vectors:

$$SD_1 = [sd^1_1, \ldots, sd^1_{i-1} sd^1_\delta, sd^1_{i+1} \ldots, sd^1_{\sqrt{n}}]$$
$$SD_2 = [sd^2_1, \ldots, sd^2_{i-1} sd^2_\theta, sd^2_{i+1} \ldots, sd^2_{\sqrt{n}}].$$

Note that all seed values except sd_δ and sd_θ are identical. Finally, $\mathcal{S}_1, \mathcal{S}_4$ receive $\langle r_\alpha, r_\beta, SD_1, b_1\rangle$, and $\mathcal{S}_2, \mathcal{S}_3$ receive $\langle r_\alpha, r_\beta, SD_2, b_2\rangle$. Here, b_1 and b_2 are called ***block vectors***, described below in detail.

Idea of decompression at the server: is to partition the table into multiple disjoint blocks, each of size $\sqrt{n}$, and execute:

$$\forall k^{th} \text{ block:}$$
$$(\mathcal{PRG}(sd^*_k[z]) + r_*[z]) \times \mathbb{A}.v_z, * \in \{\alpha, \beta\}, 1 \leq j, z \leq \sqrt{n}$$

6.2.2 Details of the Method

We provide details of the method.

1) ***Client*:** performs the following:

7. If the client wishes to fetch $x\in\{1, \sqrt{n}\}$ rows that belong to a single row or column of the matrix, the servers do not need to perform the second multiplication operation.

(*i*) ***Generates block vectors*** by organizing n tuple-ids into a matrix/grid of size $\sqrt{n}\times\sqrt{n}$. Client, then, creates two identical *block vectors* b_1 and b_2 having 0 and 1, each with $\sqrt{n}$-bits, such that an i^{th} index of b_1 and b_2 contains complemented bits, where i^{th} row of the grid contains a tuple (in cell i, j) that client wishes to fetch.
(*ii*) ***Selects two random seeds*** sd_δ ***and*** sd_θ for a PRG that produces two random vectors α_1 and α_2, each of size $\sqrt{n}$.
(*iii*) ***Generates two row vectors*** r_1 ***and*** r_2, each of size $\sqrt{n}$, by placing the row vector to the i^{th} row of the grid (which contains the desired row at the cell (i, j) to be fetched), such that $\alpha_1[j] + \alpha_2[j] + r_1[j] + r_2[j] = 1$ at the j^{th} position; otherwise, zero at all other positions.
(*iv*) ***Generates two vectors of seeds*** SD_1 ***and*** SD_2 by selecting $\sqrt{n} - 1$ random seeds, such that: SD_1=$\{sd_1, \ldots, sd_\delta, \ldots, sd_{\sqrt{n}}\}$, and SD_2=$\{sd_1, \ldots, sd_\theta, \ldots, sd_{\sqrt{n}}\}$.
(*v*) ***Sends*** $\langle b_1, r_1, r_2, SD_1\rangle$ to $\mathcal{S}_1, \mathcal{S}_4$ and $\langle b_2, r_1, r_2, SD_2\rangle$ to $\mathcal{S}_2, \mathcal{S}_3$.

2) ***Server*:** $\mathcal{S}_{z\in\{1,4\}}$ partitions the data into $\sqrt{n}$ blocks. $\mathcal{S}_1, \mathcal{S}_3$ (or $\mathcal{S}_2, \mathcal{S}_3$) keep the first (or second) share table. For each $i^{th} \in \{1, \sqrt{n}\}$ block, it selects sd_i and r_1 if the i^{th} value of the block vector is 0; otherwise, r_2. Then, servers execute the following:
$\forall$ block $i \in \{1, \sqrt{n}\}$, $\forall$ row $j \in \{1, \sqrt{n}\}$ of the block and compute: $sum_{z\in\{1,4\}\mathbb{A}} \leftarrow \sum((\text{PRG}(sd_i)[j]+r_x[j])\times\mathbb{A}.v_j)$, where PRG$(sd_i)[j]$ denotes the j^{th} values generated by PRG using seed sd_i and $x \in \{0, 1\}$ depending on the x^{th} value of block vector. The same computation is executed on other attributes of the j^{th} row to return the entire tuple.

3) ***Client*:** executes the following: $sum_{1.\mathbb{A}} - sum_{2.\mathbb{A}} - sum_{3.\mathbb{A}} + sum_{4.\mathbb{A}}$ to obtain the values of the columns. ***The same operation will be executed over other column values to obtain the complete tuple***.

6.2.3 *Discussion*

Now, we discuss the correctness, information leakage, and cost of the above method.

Correctness. Suppose, we want to fetch a value X, which is stored in additive share form: $X = (x_1+x_2) \bmod p$. We need the server executes $(X \times 1) \bmod p$, and $(Y \times 0) \bmod p$, where Y represents other values. In additive share form, the above formula can be implemented as:

$$(x_1+x_2)(\alpha_1-\alpha_2) \bmod p = (x_1\alpha_1 - x_2\alpha_2 - x_1\alpha_2 + x_2\alpha_1) \bmod p$$

Here, $\alpha_1, \alpha_2 \in \{0, 1\}$ and $\alpha_1 \geq \alpha_2$. Clearly, if we select $\alpha_1 = 1, \alpha_0$ for X and $\alpha_1 = \alpha_2$ for other values, the above formula only obtains X. Thus, the final summation at client $(sum_{1\mathbb{A}} - sum_{2\mathbb{A}} - sum_{s\mathbb{A}} + sum_{4\mathbb{A}})$ returns X. Further, since $x_1\alpha_1, x_2\alpha_2, x_1\alpha_2, x_2\alpha_1$ are calculated locally by four servers, no communication is needed among servers.

Information leakage. This method does not reveal to servers which row they return, since a server does not know both seed arrays. As servers perform identical computations on each row, it also prevents access-patterns. Further, servers return the same amount of data for each query, and this prevents volume leakage.

Cost analysis. Through iterative compressing, the size of row and block vectors can be achieved to $\mathcal{O}(\log n)$, as following the approach of [38]. Thus, the communication cost between a client and a server is $\mathcal{O}(\log n)$. Servers perform over entire data, and thus, the computation cost at the server is $\mathcal{O}(n)$, while at the client is $\mathcal{O}(\sqrt{n})$.

6.3 Comparing Two Row Retrieval Methods

Both methods offer different security guarantees and efficiency. The multiplicative-sharing-based row fetch method is information-theoretically secure, while the additive-sharing-based row fetch method is computationally secure due to using a PRG, which is computationally secure. Simply put, in the additive-sharing method, an adversary with infinite capabilities *may learn which row the client wishes to fetch*; however, the *adversary can never learn the data*. Further, due to using PRG, the additive sharing-based method is slower than another row fetch method (see Table 22 and Table 23).

7 Experimental Results

This section discusses the scalability of SeaSearch, investigates the impact of different parameters on SeaSearch, and compares SeaSearch against other systems. **Machine.** We used four `mac2.metal` AWS servers having 6 cores and 16GB RAM. We selected the same AWS machine as a combiner $\mathcal{S}_c$. Also, a similar machine is selected as a DBO/client. All such machines were located in different zones (which are connected over wide-area networks), of AWS Virginia region. **Dataset.** LineItem table of TPCH benchmark [13] with four columns $\langle$SupplyKey (SK), PartKey (PK), LineNumber (LN), OrderKey (OK)$\rangle$ is used in experiments. We created two tables with 1M and 10M cleartext rows and treated SK values as strings and others as numeric data. **Cryptographic parameters.** We set fingerprint parameter r=43 and a prime number p=100,000,007. **Code:** is written in Java and contains more than 9K lines. **Time:** is calculated by taking an average of 10 runs of the programs and shown in seconds (s).

7.1 SeaSearch Evaluation

This section investigates the following questions:

1) how much time our algorithms take to produce secret-shared tables and what will be the size of secret-shared data — Exp 1.
2) how do SeaSearch algorithms behave on different sizes of data with a single-threaded implementation — Exp 2.
3) what is the impact of parallelism over query execution – Exp 3.
4) what happens on increasing the number of columns in the conjunctive and disjunctive search — Exp 4.
5) what happens on increasing the number of rows to be fetched from servers — Exp 4.
6) how much data a client sends to a server, how much data a client fetches from a server, and how such data impacts the overall query execution time — Exp 5.
7) how much better is the idea of using a combiner — Exp 6.
8) what will happen when the number of colluding servers will increase — Exp 7.
9) how range queries behave for different range length — Exp 8.
10) how performance of multi-keyword search changes with increase in rows — Exp 9.

Exp 1: Share generation time and share data size. We create four shares tables using the algorithm given in §4. Each share table contains 9 columns: one for row-id and other columns for additive and multiplicative shares of SK, PK, LN, OK. Table 21 shows

the time to create shares and the average size of the share tables. Note that the size of a share table increases due to keeping more columns and storing each letter of a string as per the position in the dictionary.

Rows	Time for share creation and importing in MySQL	Size of a share table	Cleartext size
1M	7.2s (= 4.1 (share creation time) + 3.1 (import time))	62MB	22MB
10M	77.3s (= 35.4 + 41.9)	638MB	221MB

TABLE 21: Exp 1: Share generation time and average size of tables.

Exp 2: Query execution performance. To evaluate the query performance, we run SEASEARCH on both 1M and 10M rows using ***a single-threaded implementation*** of each entity (we discuss the impact of multiple threads later). Here, we execute conjunctive (CS) and disjunctive search (DS) over OK and PK columns. Table 22 and Table 23 show time for each operation at different entities.

Entity	String search	Number search	Conjunctive search	Disjunctive search	Row Fetch — MSR	Row Fetch — ASR
Client	0.065	0.066	0.065	0.067	0.020	0.039
Server	0.718	0.516	0.631	0.641	0.723	0.911
Total	**0.783**	**0.582**	**0.696**	**0.743**	**0.759**	**0.950**

TABLE 22: Exp 2: Time (sec) breakdown on 1M rows via 1 thread.

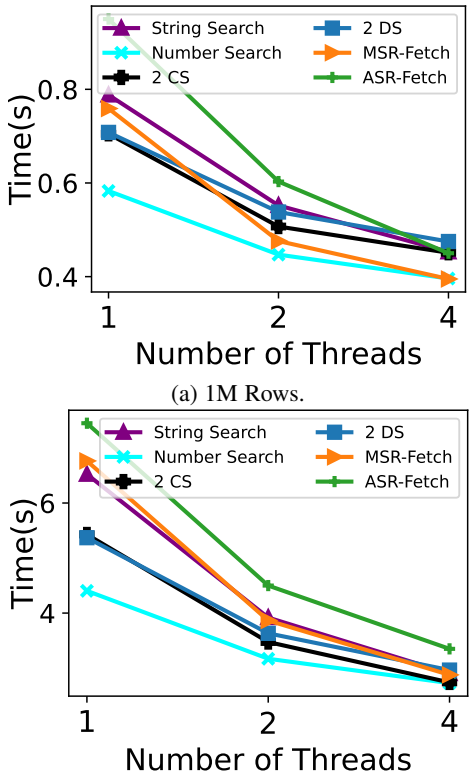
Entity	String search	Number search	Conjunctive search	Disjunctive search	Row Fetch — MSR	Row Fetch — ASR
Client	0.208	0.373	0.210	0.201	0.029	0.070
Server	6.315	4.030	5.201	5.163	6.694	8.478
Total	**6.523**	**4.403**	**5.411**	**5.364**	**6.723**	**8.548**

TABLE 23: Exp 2: Time (s) breakdown on 10M rows via 1 thread.

Maximum computation time at servers. Recall that SEASEARCH partitions a selection query into a search and fetch query. In round one for searching the qualified row-ids, SEASEARCH took at most 0.783s on 1M rows and at most 6.523s on 10M rows using one thread. In round two to fetch rows, the multiplicative-sharing-based (MSR) row fetch method took 0.759s on 1M rows and 6.723s on 10M rows, while additive sharing-based row fetch method (ASR) took 0.950s on 1M rows and 8.548s on 10M rows. We study the impact of fetching different numbers of rows later. The reason for efficient query processing is twofold: (*i*) the computation at servers is simple (just addition, multiplication, and modulo over integers), and (*ii*) servers do not need to communicate among themselves, compared to existing systems [19], [20], [30], [31], [44], [70].

Maximum computation time at a client. Computation time for the client for any operation is significantly less than 1s in the case of both 1M and 10M rows. Search queries took more time at the client compared to row fetch methods. The reason is: in search queries, the client works on either 1M or 10M numbers compared to row fetch methods that interpolate only the desired rows.

Interesting observations. The first is related to search operation: a search operation over strings takes more time than searching a number, due to computing fingerprints over additive shares of strings. (Obviously, fingerprint computation takes more time than a simple subtraction in the case of numeric data.) The second observation is related to the row fetch method: A client takes more time in ASR than MSR, since the client generates in total 4 vectors for each server in ASR compared to generating two vectors to each server in MSR. Also, a server took more time in ASR due to decompressing the vectors (via running a PRG function), compared to MSR in which servers only perform multiplication and addition.



(a) 1M Rows.

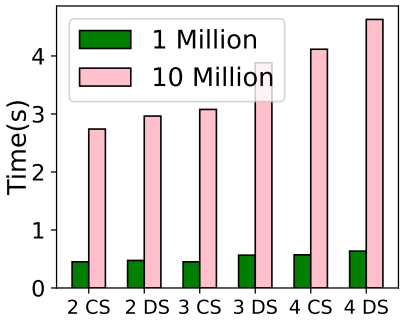
(b) 10M Rows.

Fig. 3: Exp 3: SEASEARCH performance on multi-threaded implementation at AWS. Time in seconds.

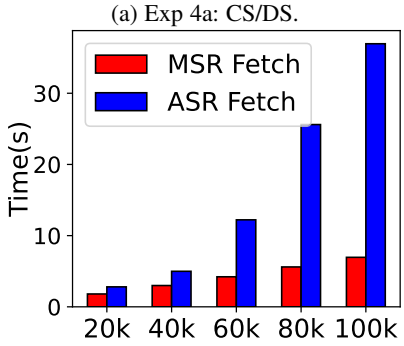
Exp 3: Impact of parallelism. SEASEARCH executes identical operations on the entire data; hence, multiple threads reduce the processing time. To inspect this, we implemented multi-threaded server programs for all algorithms. Programs create multiple blocks containing an equal number of rows, and each thread processes different parts of the data and executes the algorithm. The output of the program is kept in the memory. Figure 3 shows that as increasing the number of threads from 1 to 4, the processing time decreases. ***At 4 threads, SEASEARCH takes less than 1/2s for over 1M and less than 4s over 10M rows for executing any operation.*** Since we used only 6-core machines, increasing more than 4 threads does not help due to thrashing.

Exp 4: Impact of different parameters. We study the impact of different parameters on SEASEARCH using 4-threaded implementation of SEASEARCH, as 4-threads took the minimum time to execute a computation.

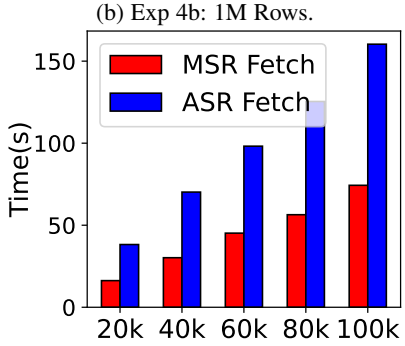
(a) The number of columns in conjunctive and disjunctive search. Figure 4a shows that as the number of columns increases from 2 to 4 in a CS search, the computation time increases slightly,



(a) Exp 4a: CS/DS.



(b) Exp 4b: 1M Rows.



(c) Exp 4b: 10M Rows.

Fig. 4: Exp 4: Impact of different parameters based on 4-threaded implementation.

as computing fingerprints over more values. The computation time also increases a bit when executing 2DS vs 3DS. However, the execution time of 4DS is significantly more than 3DS. The reason is: servers send two vectors corresponding to DS queries over $\langle$OK,SK$\rangle$ and $\langle$PK,LN$\rangle$; thus, $\mathcal{S}_c$ and the client also work on more data to obtain the final answer. Note that in 4DS, servers cannot multiply all four column values; otherwise, the secret cannot be reconstructed at the client.

(b) Impact of the number of retrieving rows. Figures 4b, 4c show that as the number of rows to be fetched increases, the computation time also increases. Two interesting observations: (*i*) the *time does not increase linearly*, as we scan/process the entire data only once for fetching multiple rows, instead of scanning/processing the entire data multiple times for each row. (*ii*) since our methods are designed to fetch $\sqrt{n}$ consecutive rows at a time, the time increases only when we fetch additional $\sqrt{n}$ rows. ASR method always took more time than MSR method, due to the decompression function at the servers.

Exp 5: Data size and the impact of communication. In our approach, servers/combiner send data to a client to answer a query.

Search algorithms: send more data from servers/combiner to the client (n integers, where n is the number of rows in the table) compared to fetch algorithms. In this case of search over 1M rows, $\mathcal{S}_c$ sends at most 7.7MB data, while 77MB data in case of 10M rows. A client sends only some numbers in any search operation.

Row fetch algorithms. In the MSR fetch method, the client sends data of size at most 12KB in the case of 1M rows and 34KB for 10M rows. The ASR fetch method requires the client to send data of size at most 14KB in the case of 1M rows and 44KB for 10M rows. In both methods, a server sends at most $\sqrt{n}$ rows of size 24KB from 1M rows and 75KB from 10M rows.

Communication cost: may impact the overall performance of SEASEARCH. We considered three different speeds of data transfer, as: slow (50MB/s), medium (100MB/s), and fast (1GB/s). The data transfer time is negligible over medium and fast speeds for both 1M and 10M datasets. In the case of slow speed, the data transfer time is also negligible for 1M data, while takes only 1s for 10M data (to transfer 77MB file). Compared to processing time, all the approaches take negligible time to transmit data, even in the case of 10M rows. Note that in all algorithms over 10M, the computation time was at least 2.7s (see Figure 3b), while the communication time is just only 1s. Thus, the communication time does not affect the overall performance of SEASEARCH.

Exp 6: Impact of the combiner $\mathcal{S}_c$. While all the above experiments include $\mathcal{S}_c$, this experiment investigates the usefulness of $\mathcal{S}_c$ by considering four cases for string search over 1M rows: (*i*) servers and the client are geographically close to each other (different zones in AWS Virginia region) and connected at 10Gbps speed, (*ii*) all servers, $\mathcal{S}_c$, and the client are in AWS Virginia region and connected at 10Gbps speed, (*iii*) all servers and $\mathcal{S}_c$ are in different zones of AWS Virginia region, while the client is at our university (NJIT) and connected at the speed of 200Mbps with $\mathcal{S}_c$, and (*iv*) servers are directly connected with the client at NJIT. The overall query processing time was 0.739s, 0.783s, 0.989s, 1.4s for the four cases respectively, while the client took 0.069s, 0.065s, 0.180s, 0.587s for the four cases respectively. This validates the purpose of using $\mathcal{S}_c$ to reduce the burden on the client when a client is far from the servers.

Exp 7: Impact of increasing the number of servers. All the above experiments consider that the servers will not collude with each other. In this experiment, we investigate when the number of colluding servers f increases. If f servers can collude with each other, then we need to create $f+1$ additive or multiplicative shares. Thus, a group of f servers cannot recover the secret. However, if we consider f Byzantine servers, which our paper does not deal with, then we need $2f+1$ servers.

We experimented with the first case for a simple search. Here, increasing the number of servers does not impact the computation time at the servers, since servers work in parallel. Also, the computation time at the client does not change, as the client works only on one vector. However, the combiner S_c performs operations over more values, and hence, the computation time at S_c, (as well as, the overall computation time) increase slightly. Table 24 shows the time at $\mathcal{S}_c$ as increasing the number of colluding servers f and the total computation time for a simple search.

Exp 8: Range queries. As we increase the length of a range, the computation time also increases; see Table 25. We did experiments for different range lengths of 50, 100, 500, and 1000. Recall

#servers	$f=0$	$f=1$	$f=2$	$f=4$	$f=8$
Time at $\mathcal{S}_c$	0.066	0.071	0.076	0.081	0.101
Total time	0.582	0.587	0.592	0.597	0.617

TABLE 24: Exp 7: Computation time at $\mathcal{S}_c$ with f colluding servers.

that range queries create different blocks at servers; we vary the number of blocks to 10 and 100. However, as increasing blocks, the computation time does not increase significantly (only differs in ms). The reason is: as we increase the number of blocks servers do the identical computation, while the computation at the announcer increases a little bit. In this experiment, A-Range refers to the method of §5.6.1, while pA-range refers to the method of §5.6.2.

	pA-Range		A-Range	
Length	10Blo	100Blo	10Blo	100Blo
50	4.852	5.463	1.835	2.117
100	5.375	6.081	2.139	2.815
500	7.315	9.763	4.821	5.224
1,000	9.505	14.493	7.367	7.511

TABLE 25: Exp 8: Impact of range conditions on 10M rows using 4 threads (time in seconds).

Exp 9: Multi-keyword Search. We performed multi-keyword search (§5.5) for ten keywords and created ten blocks. String-based multi-keyword search took 1.479s for 1M rows and 10.761s for 10M rows on a single thread. Number-based multi-keyword search took 0.679s for 1M rows and 3.624s for 10M rows on a single thread. We observe that multi-keyword search for numeric columns takes less time as compared to multi-keyword search on string columns, due to the fingerprint operation performed over string columns. In comparison to single keyword search, Table 22 and Table 23, multi-keyword search takes more time for both numeric and string search; however, this time does not scale linearly as we search multiple keywords in one round of communication.

7.2 SeaSearch vs Other Systems

SeaSearch offers information-theoretic security; thus, we compare SeaSearch against systems offering the same level of security. While multiple information-theoretically secure systems are developed by industries (*e.g.*, Sharemind [20], SPDZ [30]-based Systems), they are not freely available. To give a perspective on query execution time and compare SeaSearch against those systems, Table 1 provides experimental results given in the respected papers. Below, we compare SeaSearch against the following: trivial download strategy and additive sharing-based Jana [19], Waldo [33], Ciphercore [15]. Table 4 shows such results. Important to note that ***the existing secret-sharing systems do not support the large data and take more time compared to*** **SeaSearch**.

Download methods. We compare SeaSearch against three download strategies (given in §1). Overall, SeaSearch outperforms such methods.

Jana [19]: supports only selection queries over additive shares in a single round of communication between a client and a server. Also, Jana requires servers to communicate among themselves to execute a query. Jana converts all non-desired rows (*i.e.*, rows not containing the keywords) into zero in additive share form and returns the entire database to the client that filters the desired rows. Jana took more than 10 minutes to create shares of 1M rows. For executing a selection query, Jana took ≈450s.

Waldo [33]: allows a client to know the presence/absence of a query keyword over additive shares. However, it does not allow us to know the row-id where the keyword exists. Waldo took ≈12s for searching a keyword.

Ciphercore [15]: supports only search operation using `equality operator` in a single round of communication between a client and a server, while requiring servers to communicate among themselves to execute a query. Ciphercore returns a vector containing 1 or 0 to the client, where 1 shows which row contains the query keyword. However, the current version of Ciphercore does not support operations to fetch the desired row. Furthermore, since Ciphercore is proprietary software, the current code does not allow us to find separate times to create shares and time to execute a query. In other words, the current code requires creating the share of the entire data before executing each query. Using one thread, Ciphercore took more than 1 minute for both share creation and executing a search query over 1M rows, while SeaSearch took at most 7.988s (7.2s to create shares and 0.788s for a search query).

S^{3}ORAM [44]: is a multiplicative sharing-based method to execute a search over only a single column via an ORAM-type index. S^{3}ORAM inherits all the weaknesses of ORAM, as discussed in §1.2. Also, S^{3}ORAM does not support conjunctive/disjunctive search. The current code allows searching only *unique* random numbers and incurs the high space overhead by storing twice the amount of input numbers. Except random numbers, the current code does not import other data. To provide a perspective on query execution time, we provide experimental results (taken from the paper) of S^{3}ORAM in Table 1.

8 Related Work

Secret-Sharing-based solutions. Additive [20] and multiplicative [67] are the two famous secret-sharing techniques. Such techniques perform addition over shares efficiently locally at servers, (*i.e.*, without communicating with other servers), while the multiplication of shares requires communication among servers [18]. Multiplicative shares can multiply shares locally at servers, if we have enough shares. Sharemind [20], SPDZ [30], [31], Jana [19], Conclave [70], and Waldo [33] use additive shares. PDAS [69], and [34], [73] use multiplicative shares. However, such techniques suffer from either query inefficiency and/or information leakage via access-patterns and/or volume and/or use a trusted party as in Conclave, as we have discussed in §1. Table 1 compares such techniques also. In contrast, SeaSearch offers highly efficient query processing using both additive and multiplicative shares, as well as also prevents leakages from both access-patterns and volume. Also, SeaSearch does not use any trusted party.

Information leakage via access-patterns. The impact of revealing access-patterns on encrypted data was investigated in the novel work [46] that lead to multiple works in the same

direction [24], [35], [42], [51], [53], [55], [60], [61]. To overcome leakage from access-patterns, ORAM [40], [41], [63] and their improved version known as PathORAM [68] was developed. Such solutions have asymptotic complexity of polylogarithmic in the index size. However, all such solutions have multiple problems, as mentioned in §1. S3ORAM [44] provides ORAM-type index for secret-shares. However, S3ORAM [44] does not support conjunctive/disjunctive/range search. Furthermore, S3ORAM [44] also inherits all disadvantages of the standard ORAM. PIR, DPF, and FSS also hide access-patterns. Splinter [71] is the first system based on FSS. SEASEARCH provides two techniques for row fetch while hiding access-patterns: one is information-theoretically secure and another is based on DPF for additive shares.

Information leakage via volume. [65] showed that even when hiding access-patterns, an adversary can learn based only on volume. Recently, several techniques [17], [47], [64], [66] have been developed to hide volume. While [47], [64] are data-independent volume-hiding techniques, all such techniques only deal with encrypted key-value data only. Furthermore, all these techniques incur significant storage overhead (by storing ciphertext that is at least twice the actual data [17], [47], [64]) and show inefficient query execution (by fetching data that is twice the size of the maximum result).

9 Extending to Group-by Operations

In executing group-by queries, the client needs to know the name of the groups, and then, based on that, the client can ask the servers to execute group-by queries. SEASEARCH supports group-by count, sum, maximum, minimum, and median queries. As will be clear below, the communication cost between a server and a client to know the final answer to a query is minimum in group-by-count and sum queries.

Group-by count: requires only one round of communication. Once the client knows the group names, client can execute single keyword search operation (§5.1) to know the answer to the count query. Particularly, the client can search for each group name using the approach of §5.1 or §5.5. Recall that we compute fingerprints over strings to search for a keyword, and it may result in false positives with a negligible probability. To completely avoid false positive probability, which is essential in group-by operations, DBO can map each group to a random number, and this can be done with the help of pseudo pseudo-random number generator.

Group-by sum: requires two rounds of communication between a server and a client. In the first round, the client executes keyword search operation (§5.1 or §5.5) to know the row-ids matching the group name. Then, in the second round, the client sends a vector containing 0 and 1 in multiplicative share form to three servers. Each server multiplies the i^{th} number of the vector with the i^{th} value of the desired column on which the client wishes to compute the sum. After that, the server adds all values of the column and sends the answer to the client. On receiving an answer to the query, the client performs Lagrange interpolation to know the final answer.

Group-by max/min/median: requires two rounds of communication between a server and a client. In the first round, the client executes the keyword search operation (§5.1 or §5.5) to know the row-ids matching the group name. Then, in the second round, the client executes PRG row fetch method to know the values of the desired column that contains the group name. On receiving all the values of the column that contains the desired group name, the client locally finds the maximum/minimum/median.

10 Conclusion

We develop SEASEARCH — efficient and scalable techniques for selection queries, based on both additive and multiplicative secret-sharing. SEASEARCH does not reveal information from ciphertext and query execution via both access-patterns and volume/output-size, simultaneously. SEASEARCH uses fingerprints to perform search operations over the shares. The fingerprints avoid communication among servers during query execution, and this brings in efficiency, as justified by experiments.

References

[1] Biometrics and blockchains: the Horcrux protocol [part 3]. Available at: https://tinyurl.com/2c2jpmfd.
[2] Binance Moved $204 million Worth Of ETH For A Fee Of 6 Cents. Available at: https://tinyurl.com/yc3bn8xp.
[3] Thailand's Democrat Party Holds First Ever Election Vote with Blockchain Technology. Available at: https://tinyurl.com/y26rztj7.
[4] Coinbase Moves $5Billion Worth of Crypto to Kick-Start its new Digital Storage System. Available at: http://tinyurl.com/bddh2znk.
[5] Multicloud. Available at: https://www.ibm.com/cloud/learn/multicloud.
[6] Multi-cloud mature organizations are 6.3 times more likely to go to market and succeed before their competition. Here's why. Available at: https://www.geektime.com/multi-cloud-maturity-report-seagate/.
[7] More and more companies are spreading their data over public clouds. Available at: https://tinyurl.com/46fph54z.
[8] Multi-Cloud Data Solutions for Today (and Tomorrow). Available at: http://tinyurl.com/ym7hkn8v.
[9] How Many Companies Use Cloud Computing in 2022? All You Need To Know. Available at: https://tinyurl.com/2p983aau.
[10] Jana: Private Data as a Service. Available at: https://galois.com/project/jana-private-data-as-a-service/.
[11] Stealth Pulsar, available at:http://www.stealthsoftwareinc.com/.
[12] Cybernetica's Sharemind. Available at: https://sharemind.cyber.ee/secure-computing-platform/.
[13] TPC-H. Available at: https://www.tpc.org/tpch/.
[14] Code, data, and the full version of the paper: https://drive.google.com/drive/folders/1jTdLkRn2qD_Nix4_CbjR8y8rt3x2SyXg?usp=sharing.
[15] Ciphercore GitHub. Available at: http://tinyurl.com/3dw62y4c.
[16] I. Ahmad et al. Coeus: A system for oblivious document ranking and retrieval. In *SOSP*, pages 672–690, 2021.
[17] G. Amjad et al. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *PETS*, 2023(1):417–436, 2023.
[18] T. Araki et al. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*, pages 805–817, 2016.
[19] D. W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.
[20] D. Bogdanov et al. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.
[21] E. Boyle et al. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.
[22] D. Breslauer et al. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4), 2014.
[23] M. Burkhart et al. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security*, pages 223–240, 2010.
[24] D. Cash et al. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.
[25] S. G. Choi et al. Efficient three-party computation from cut-and-choose. In *CRYPTO*, pages 513–530, 2014.
[26] B. Chor et al. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, page 3, 1998.
[27] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, nov 1998.
[28] R. M. Corless et al. A graduate introduction to numerical methods. *AMC*, 10:12, 2013.
[29] R. Cramer et al. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[30] I. Damgård et al. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, volume 7417, pages 643–662, 2012.

[31] I. Damgård et al. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, pages 1–18, 2013.

[32] E. Dauterman et al. DORY: an encrypted search system with distributed trust. In *OSDI*, pages 1101–1119, 2020.

[33] E. Dauterman et al. Waldo: A private time-series database from function secret sharing. In *IEEE SP*, pages 2450–2468, 2022.

[34] F. Emekçi et al. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.

[35] F. Falzon et al. Attacks on encrypted range search schemes in multiple dimensions. *IACR Cryptol. ePrint Arch.*, page 90, 2022.

[36] J. Frankle et al. Practical accountability of secret processes. In *USENIX Security Symposium*, pages 657–674, 2018.

[37] M. J. Freedman et al. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324, 2005.

[38] N. Gilboa et al. Distributed point functions and their applications. In *EUROCRYPT*, volume 8441, pages 640–658, 2014.

[39] I. Goldberg. Improving the robustness of private information retrieval. In *IEEE SP*, pages 131–148, 2007.

[40] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194. ACM, 1987.

[41] O. Goldreich et al. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[42] P. Grubbs et al. Leakage-abuse attacks against order-revealing encryption. In *IEEE SP*, pages 655–672, 2017.

[43] P. Grubbs et al. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE SP*, pages 1067–1083, 2019.

[44] T. Hoang et al. S^3oram: A computation-efficient and constant client bandwidth blowup ORAM with shamir secret sharing. In *CCS*, pages 491–505, 2017.

[45] R. Inbar et al. Efficient scalable multiparty private set-intersection via garbled bloom filters. In *SCN*, pages 235–252, 2018.

[46] M. S. Islam et al. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[47] S. Kamara et al. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.

[48] R. M. Karp et al. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[49] J. Katz et al. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, second edition, 2014. See Section 5.4.1 for birthday paradox.

[50] J. Katz et al. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, second edition, 2014. See Section 2.2 for one-time pad.

[51] G. Kellaris et al. Generic attacks on secure outsourced databases. In *CCS*, pages 1329–1340, 2016.

[52] J. Liagouris et al. Secrecy: Secure collaborative analytics on secret-shared data. *CoRR*, abs/2102.01048, 2021.

[53] C. Liu et al. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.

[54] E. A. Markatou et al. Full database reconstruction with access and search pattern leakage. In *ISC*, volume 11723, pages 25–43, 2019.

[55] E. A. Markatou et al. Full database reconstruction with access and search pattern leakage. In *International Conference on Information Security*, pages 25–43, 2019.

[56] S. Mehrotra et al. PANDA: partitioned data security on outsourced sensitive and non-sensitive data. *ACM Trans. Manag. Inf. Syst.*, 11(4):23:1–23:41, 2020.

[57] C. A. Melchor et al. XPIR : Private information retrieval for everyone. *Proc. Priv. Enhancing Technol.*, 2016(2):155–174, 2016.

[58] P. Mohassel et al. Fast database joins and PSI for secret shared data. In *CCS*, pages 1271–1287, 2020.

[59] M. Naor et al. Access control and signatures via quorum secret sharing. *IEEE Trans. Parallel Distributed Syst.*, 9(9):909–922, 1998.

[60] M. Naveed. The fallacy of composition of oblivious RAM and searchable encryption. *IACR Cryptol. ePrint Arch.*, page 668, 2015.

[61] M. Naveed et al. Inference attacks on property-preserving encrypted databases. In *CCS*, pages 644–655, 2015.

[62] C. Orlandi. Is multiparty computation any good in practice? In *ICASSP*, pages 5848–5851, 2011.

[63] R. Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, MIT, Cambridge, MA, USA, 1992.

[64] S. Patel et al. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *CCS*, pages 79–93, 2019.

[65] R. Poddar et al. Practical volume-based attacks on encrypted databases. In *IEEE EuroS&P*, pages 354–369, 2020.

[66] K. Ren et al. Hybridx: New hybrid index for volume-hiding range queries in data outsourcing services. In *ICDCS*, pages 23–33, 2020.

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

[68] E. Stefanov et al. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.

[69] B. Thompson et al. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *PETS*, pages 185–201, 2009.

[70] N. Volgushev et al. Conclave: secure multi-party computation on big data. In *EuroSys*, pages 3:1–3:18, 2019.

[71] F. Wang et al. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313, 2017.

[72] H. Wang et al. On secret reconstruction in secret sharing schemes. *IEEE Transactions on Information Theory*, 54(1):473–480, 2008.

[73] T. Xiang et al. Processing secure, verifiable and efficient SQL over outsourced database. *Inf. Sci.*, 348:163–178, 2016.

[74] E. Zhang et al. Efficient multi-party private set intersection against malicious adversaries. In *CCSW*, page 93–104, 2019.

APPENDIX A
SECURITY PROOF OUTLINE

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

Theorem 1. *For an adversary $\mathcal{A}$ executing a server protocol π over any input secret-share relation $\mathbb{R}$ of $n{>}1$ rows and $m{>}0$ columns and for any query predicates qp, qp', the protocol π is secure, iff the following condition holds (where $\mathcal{L}$ could be either access-pattern leakage or volume leakage):*

$$\mathcal{A}_{view}(\pi, qp, \mathbb{R}, \mathcal{L}) = \mathcal{A}_{view}(\pi, qp', \mathbb{R}, \mathcal{L}) \blacksquare$$

We can argue that if the adversarial server can distinguish two input queries, then either SEASEARCH does not create shares randomly or SEASEARCH does not provide query privacy.

In order to show that the adversarial server can never know the exact query value, we consider two instances of the databases, 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. Of course, D_1 and D_2 contain n rows and m columns. Here, we show that if the adversary can distinguish the single different value in D_1 and D_2, she can break SEASEARCH. In this setting, the server executes the input queries on D_1 and D_2.

The adversary cannot distinguish that D_1 and D_2 are identical or different, due to randomness in creating shares by following the algorithm given in §4. Note that if the DBO uses only one polynomial (*i.e.*, a weak cryptographic plan) and the same additive shares for a value, then the adversary can find which value is the only single value of D_1 that is different from values of D_2. Moreover, it reveals frequency count of values. However, the share creation algorithm will never create the same shares for the same values.

Now assume the queries for the value v_1 and v_2 that will be mapped to secret-shared queries, $q_{v1}(D_1)$ and $q_{v1}(D_2)$, respectively. These queries can be executed on either or both databases. Further, assume that $q_{v1}(D_1)$ and $q_{v2}(D_2)$ are identical in share form. Hence, the adversary will consider both of them as an identical query, while they are for different queries. Hence, the adversary cannot distinguish two queries. Now, assume that $q_{v1}(D_1)$ and $q_{v2}(D_2)$ are different in share form, and here the adversary's objective is to deduce which tuples of relations satisfy the query or not. If the adversary cannot know which tuple is satisfying the query, the adversary cannot distinguish two queries, as well as, the two datasets. Recall that all the algorithms access the entire database and perform the same operations on each row. Further, the servers send the same amount of data regardless of the query predicate. Thus, the adversary cannot distinguish two datasets or two queries. ■

Theorem 2. *For any given secret-shared relation at the servers, for any query predicate qp, and for any real client, say C, there exists a probabilistic polynomial time (PPT) client C' in the* ideal execution*, such that the outputs to C and C' for the query predicate qp on the secret-shared relation are identical.* ■

Now, we show how server privacy is maintained. In order to show that the client will learn only the answer to the query, we consider two instances of the datasets, as follows: D_1 and D_2, where D_1 and D_2 hold n rows and m columns. A single value v_1 appears in both D_1 and D_2. Except v_1, all the other rows in D_1 and D_2 can contain the same or different values. Here, we show that if the client executes a query for v_1, the client will not learn any additional information other than the output of the query. In this setting, the server executes the input queries on both D_1 and/or D_2.

Note that the server cannot distinguish between D_1 and D_2. Further note that in response to a query, the client will surely obtain only the correct answer in the ideal execution of the protocol.

The query for v_1 is mapped to secret-shared queries, $q_{v1}(D_1)$ and $q_{v1}(D_2)$, respectively, for database D_1 and D_2. Note that the server cannot distinguish between two queries. Now, servers will return all the row-ids matching the query in the case of the search operation (otherwise, return the rows during the fetch operation). Now, the task of the client is to find out which database is used by the server to answer a query. The server can execute both queries on the same database or a different one.

Now, consider two cases: both queries are executed on the same database, and before returning the row-ids to the client, the servers permute the answer; and both queries are executed on different databases, and answers are returned to the client. The client here cannot know which database is used to answer the query, since the answer to the query will return the same number of matching rows in both cases. To break the technique, the client needs to know $seed_s$, which is only known to the server.

This shows that the client will not receive any additional information other than the answer to queries, such that the client is able to distinguish the databases, and furthermore, the answer will not allow the client to learn the same number of qualified rows as they can be returned in the ideal execution of the protocols. ■