Information-Theoretically Secure and Highly Efficient Search and Row Retrieval

Shantanu Sharma
NJIT, USA
shantanu.sharma@njit.edu

Yin Li
DGUT, China
yunfeiyangli@gmail.com

Sharad Mehrotra
UCI, USA
sharad@ics.uci.edu

Nisha Panwar
Augusta University, USA
npanwar@augusta.edu

Komal Kumari
NJIT, USA
kk675@njit.edu

Swagnik Roychoudhury
NYU, USA
sr6474@nyu.edu

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 S^2, 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 S^2 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. S^2 does not require communication among servers storing shares and does not reveal any information to an adversary based on access-patterns and volume.

PVLDB Reference Format:
Shantanu Sharma, Yin Li, Sharad Mehrotra, Nisha Panwar, Komal Kumari, and Swagnik Roychoudhury. Information-Theoretically Secure and Highly Efficient Search and Row Retrieval. PVLDB, 16(10): 2391 - 2403, 2023.
doi:10.14778/3603581.3603582

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SecretDeB/S2-VLDB-2023.

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. The 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 [72], PDAS [71], Obscure [44], and [34, 74]. Existing systems (by both academia and industry), 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 [71] and [34, 74] reveal both access-patterns and volume. Pulsar [11], Conclave [72], and Obscure [44] reveal volume. However, *efficient* SS-based systems do not prevent both

[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!

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603582

Table 1: Comparison of existing secure systems against S^2. 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 [74] experimental comparison. (*vi*) ‡‡: Does not mention the number of rows. (*vii*) ⋄: Numbers are taken from Conclave [72] experimental comparison. (*viii*) ⊖: Numbers from [56]. (*ix*) †: **Conclave [72] uses a trusted party** to support SPJ over multi-party settings, and thus, we do not include experimental numbers.

Papers	[34]	S3ORAM [45]	Obscure [44]	Sepia [23]	Sharemind [20]	SPDZ [30]	Jana [19]	Conclave [72]	S^2
Technique	SSS	SSS	SSS	SSS	Additive	Additive	Additive	Additive	**Additive + SSS**
Communication between servers	No	Yes	No	Yes	Yes	Yes	Yes	Yes	**No**
Distribution/frequency leakage from ciphertext	No	No	No	No	No	No	No	No	**No**
Access-pattern leakage from query execution	Yes	No	No	No	No	No	No	Yes	**No**
Volume leakage from query execution	Yes	No	No	No	No	No	No	Yes	**No**
Supported operators	SPJ	1 keyword fetch	Aggregation & fetch	Compare/ equality	SPJ	SPJ	Selection	†	**Complex search & fetch**
Computational Complexity of selection	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Index support	Yes	Yes	No	No	No	No	No	No	**No**
Support for dynamic data	Yes	No	Yes	N/A	Yes	Yes	Yes	Yes	**Yes**
Experimental results: Time *using one thread* (data)	**0.07s∗ (150K rows)**	**7.3s (on 40GB)‡‡**	**600s (on 6M rows)**	**N/A**	**>10m (on 3M rows)⋄**	**10s (on 1000 rows) ⊖**	**450s (on 1M rows)**	†	**1.502s on 1M rows**

the leakages, simultaneously. Prior work has shown the importance of preventing both leakages [21, 24, 43, 47, 51, 53, 54, 60, 64, 66].

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 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 [72]. The two main reasons for 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. For example, in Secrecy [52] for searching keywords over m columns and n rows can require $O(m\ell)$ communication rounds where ℓ is the maximum length of a word, and the total amount of information flow among servers/client will be at most $(24m\ell + 1)n$ bits, (depending on the data type of m columns). To have efficient query execution, optimizations can come in two ways: reducing the number of rounds among servers and the amount of information flow among servers. (As will become clear soon, our approach requires $n\kappa$ bits to be sent from each of the two servers to the client for a simple keyword or conjunctive search or at most $nm\kappa/3$ bits for a disjunctive search, where κ refers to bits requires by a data type that could be `integer` or `double` in our proposed approach.)

This paper describes ***efficient, scalable, and information-theoretically secure techniques for selection queries***, entitled $\boldsymbol{S^2}$, that prevents information leakages from both access-patterns and volume. S^2 does not require servers (storing secret-shares) to communicate among themselves before/during/after computations. S^2 offers: (*i*) ***query privacy***: indistinguishability of queries by an adversarial server, (*ii*) ***data privacy***: not revealing to an adversarial server anything (*e.g.*, data distribution and ordering) about input/intermediate/output data, and (*iii*) ***server privacy***: not revealing to queriers/clients anything other than answers to queries.

Before describing, how S^2 achieves the goal of efficient, scalable, and secure search, we first briefly discuss secret-sharing techniques.

1.1 Background

S^2 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 they collect all c shares. To reconstruct S, the secret owner collects all the 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 [69], 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 also 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 each multiplication increases the polynomials' degree.

Fingerprint. The fingerprint (function) was proposed for string matching over cleartext [49]. 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

Table 2: S^2 performance (sec) on 1M rows using 1 & 4 threads.

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: Comparing different systems against S^2 on 1M rows.

Method	Total query time	Speedup	Operation support
DL additive shares	4.9s (1.2s to DL, 1.5s to add, 2s to L, 0.2s to Q)	≈3x	Any
DL one-time pad	4.6s (1.2s to DL 1.2s to XOR 2.0s to load 0.2s to Q)	≈3x	Any
Jana	≈450s	≈300x	Selection query
Waldo	≈12s	≈7x	Absence/presence of a keyword
Obscure	≈150s	≈99x	Selection query
S^2	**≈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.

identical strings always generate the same fingerprint. As fingerprints execute a modular operation, the probability of false positives exists. S^2 ***executes fingerprints over additive shares for string search and conjunctive search*** (see §5.1.1). Particularly, S^2 computes fingerprints over the shares of the database and the shares of a query keyword. A false positive will be produced if the fingerprint computed over a value in the share form of the database is equal to the fingerprint computed over the share of the query under the modulus operation, while the cleartext value in the database is not identical to the cleartext query keyword. The probability of this is $1/p$. Since there are n rows in the database, the probability of a single collision is n/p. For a suitably large p (*e.g.*, $p \gg n$), the probability of collision is negligible. In our experiments with $r = 43$ and $p = 100{,}000{,}007$, we get zero false positives for queries.

1.2 Summary of S^2

S^2, primarily, ***supports selection queries containing conjunctive, disjunctive, and range predicates. Also, S^2 can offer sum and group-by sum queries,*** which are provided in the full version.

To execute such queries, S^2 executes two rounds of communication between servers and a client. In the first round, S^2 finds the row ids that satisfy the query predicate, and then in the second round, fetches all the qualified rows (or executes addition operation for answering sum queries). S^2 uses additive shares for single/multiple keyword search, conjunctive search, and search involving range conditions, while uses multiplicative shares for disjunctive search. Importantly, S^2 *does not require servers (which store secret-shares) to communicate among themselves before/during/after computations.* All supported operations by S^2 prevent both access-patterns and volume leakage, simultaneously.

Each round of S^2 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 S^2 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 S^2 at the cloud must be private. A trivial and impractical approach to address this challenge is to download a complete copy of the entire secret-shared 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 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 reveals 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 [49], 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.

2. Efficient and Oblivious Row Retrieval. Once we know the row ids that have the query keyword in round one of S^2, 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, 70]. 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, and range queries, no efficient support for dynamic data, and harder to support multiple clients, as also argued in [32]. Another solution is to use PIR or optimized versions of PIR, known as Distributed Point Function (DPF) [38] and Function Secret Sharing (FSS) [22]. S^2 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 $O(\sqrt{n})$ 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 $O(\log n)$ 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 $O(\sqrt{n})$ rows in the same round with the same communication and computational cost. Importantly, our methods are significantly better than existing work [19, 20, 44, 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 S^2.

3. S^2 performance. We implemented S^2 in Java, and the code contains more than 9,000 lines. We set up S^2 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), S^2 took at most 0.475s on 1M rows and 2.964s on 10M rows using four-threaded implementation. In *round two to fetch row*, S^2 took 0.395s using multiplicative shares and 0.452s using additive shares on 1M rows

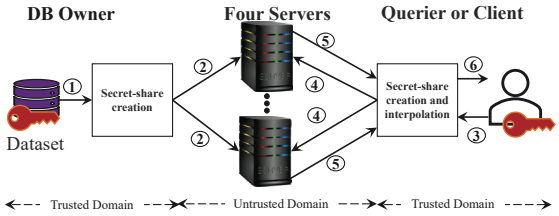


Figure 1: The model.

using four-threads. On 10M rows, in round two, S^2 took 6.765s using multiplicative shares and 3.357s using additive shares using four threads. Table 2 provides computation time over 1M rows using one/four threads, and other experiments are given in §7.

4. S^2 vs other approaches, and the reason for using fingerprints. The following two simpler approaches could support the same class of operators as S^2: (*i*) *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. (*ii*) *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.

These two approaches are information-theoretically secure, offer the same security as S^2 offers, do not require dependence on fingerprints, and do not require any communication among the cloud servers. These approaches, however, incur a huge communication cost in downloading the secret-share data and the computational cost at the client to execute a query. Compared to these approaches, S^2 first finds the desired row ids, and then, fetches those rows.[2] It is important to note, a fingerprint function, in S^2, 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 3 compares S^2 against the above three approaches and also to other approaches.

Another simple approach can store encrypted data at the cloud, and the client downloads the entire encrypted data, decrypts it, and loads the cleartext data into a DBMS to execute a query. However, this approach is not information-theoretically secure.

Code, data, and full version of the paper: is given in [14].

2 PRELIMINARY

This section provides an overview entities involved in S^2, 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 4 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] S^2 could use a single round of communication between a server and the client by downloading all projection columns of a selection query and downloading the results of the selection predicate. This strategy could be better if queries are not selective and many rows satisfy the query. In contrast, the two-round strategy as used in S^2 works well when queries are highly selective.

Table 4: Frequently used notations in this paper.

Notations	Meaning
R and A	A relation/table in cleartext and the attribute/column A of the table 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 $\mathcal{PRG}$

(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 be 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, 67]. Servers establish a secure communication link with the DBO and the 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 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 do 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.

Table 5: Parameters known to different entities.

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

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, 47, 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.

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.)

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 S^2, 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', the protocol π is secure, iff the following condition holds (where $\mathcal{L}$ is as 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.* ■

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 full version [14].

Later sections will discuss in detail the security of different operators offered by S^2. Informally, S^2 uses secret-sharing techniques over cleartext to produce different 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. S^2 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, S^2 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.

Table 6: Techniques and servers used for different operations.

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

2.3 Evaluation Parameters

S^2 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 always zero***, since S^2 does not require communication among servers.

Briefly, in S^2, each individual operator takes only one round of communication between the desired two entities. The maximum communication cost from a server to a client is $O(n)$ 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 $O(\sqrt{n})$. The scan cost at servers and a client is one. The computation cost varies for different operations and is discussed with each operator.

3 S^2 AT THE HIGH LEVEL

S^2 consists of three entities: a trusted DBO, four untrusted servers, and clients; see Figure 1. S^2 provides *oblivious* algorithms for search and row retrieval. At the abstract level, data processing in S^2 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.

Table 7: An input cleartext `Patient` table.

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

Table 8: PATIENT$_1$.

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 9: PATIENT$_2$.

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

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. Clients determine the final answer to search queries, *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.

4 DATA OUTSOURCING IN S^2

This section explains how S^2 uses different secret-sharing techniques on a table/relation and outsources them. To state it briefly, *S^2 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 equalize lengths. 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, based 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 S_1,S_3 and $\mathbb{R}_2$ to S_2,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. 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 to know row-ids, satisfing 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. S^2 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,

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

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 row-ids (containing the query keyword), which will help to fetch the rows.[3]

5.1.2 ***Example.*** A client wants to know whether Tables 8, 9 contain "Jo" or not in the $\mathbb{A}$(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: $(5\times2+5\times2^2) \bmod 17 = 13$ and $(5\times2+10\times2^2) \bmod 17 = 16$. 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: $(14 + 7) \bmod 17 = 4$ $(11 + 15) \bmod 17 = 9$ $(6 + 11) \bmod 17 = 0$ $(16 + 16) \bmod 17 = 15$
The vector $\langle 4, 9, 0, 15\rangle$ is compared against $\mathcal{PRG}(seed_c)$=[4, 6, 1, 2], and only the first position matches. This shows that the first row of the data contains the query keyword Jo. ■

[3] 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.

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

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

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	$10\times2+10\times2^2 \bmod 17 = 9$	$(9-16) \bmod 17 = 10$	$(10\times5) \bmod 17 = 16$

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,

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

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) 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 $O(n)$, 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 $O(kn)$, 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.

Table 12: Search computation at $\mathcal{S}_1$.

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

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	$10{\times}2{+}10{\times}2^2{+}2{\times}2^3 \bmod 17{=}8$	$(8-15) \bmod 17{=}10$	$(10{\times}5) \bmod 17{=}16$

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: $\langle 5, 5, 2\rangle$ $\langle 5, 10, 2\rangle$. Finally, fingerprints are created: $(5{\times}2+5{\times}2^2+2{\times}2^3) \bmod 17{=}12$ 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 $\mathcal{S}_2$: $(13{+}8) \bmod 17{=}4$ $(11{+}6) \bmod 17{=}0$, $(2{+}7) \bmod 17{=}9$, $(16{+}16) \bmod 17{=}15$ 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 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 does not know $seed_s$, the client only learns rows satisfying the query, nothing else.

Cost analysis. The computation cost at the server is $O(n)$. 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 $\mathcal{PRG}(seed_s)[i]$, adds $\mathcal{PRG}(seed_c)[i]$, and sends $\lceil k/3 \rceil$ 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 $\mathcal{PRG}(seed_c)[i]$. If it matches, then the i^{th} row satisfies the disjunctive query. (Note that since S^2 uses at most four servers, the client can interpolate the shares of four servers to recover the answer.)

Information leakage discussion and cost analysis: are presented in the full version [14].

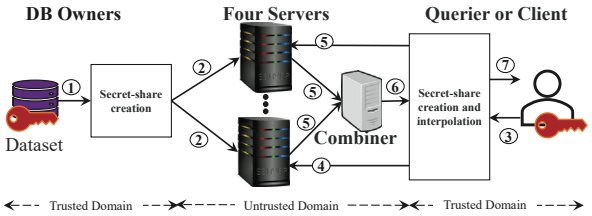


Figure 2: The model with combiner.

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, with a limited network connection speed, and/or hold a weaker machine. Here, the following can happen: (*i*) the processing time at the client will increase, and/or (*ii*) 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. 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 cleartext data. 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 [16, 25, 46, 73, 75].

Method. This method is straightforward. Now, servers send the output of the computation to $\mathcal{S}_c$ instead of sending it to clients. Importantly, servers ***must use*** $\mathcal{PRG}(seed_c)$ in Step 2 of the respective algorithms. In ***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 servers satisfies the query. Note that now $\mathcal{S}_c$ sends only n numbers to the client that *does not* need to perform modular addition. In ***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 $\mathcal{PRG}$. Until now, one can check the necessity of $\mathcal{PRG}(seed_c)$: if server does not add such random numbers, then $\mathcal{S}_c$ will learn which rows and how many rows satisfy the 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, 44]). 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

6.1.1 ***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 $r = c = \sqrt{n}$.[4] 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.

						Row id
0	0	⋯	0	⋯	0	1
⋮	⋮	⋱	⋮	⋮	⋮	⋮
0	0	⋯	1	⋯	0	*i*
⋮	⋮	⋱	⋮	⋮	⋮	⋮
0	0	⋯	0	⋯	0	$\sqrt{n}$
1	2	⋯	*j*	⋯	$\sqrt{n}$	Column ids

S^2 implements exactly the same. Particularly, client sends r_1 and r_2 vectors in multiplicative share form, and each of the four servers implements exactly the same idea over multiplicative shares.

6.1.2 ***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

[4] 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 $n = x' \times y'$.

the output of each attribute across $\sqrt{n}$ tuples, if the client wants to fetch only $(i, j)^{th}$ row.[5] 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 $O(n)$, while at the client is $O(x)$, 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 $O(\sqrt{n})$. 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] provides a trivial PIR scheme to obliviously obtain one bit from two servers. This method can be extended to fetch $i^{th}_{i\in\{1,n\}}$ additive share using four servers: 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*. 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, $\forall i\in\{1, n\}\setminus\{k\}$ and $r_1[k]-r_2[k]$=1. This method incurs high communication cost of n-bits to fetch a single row. Below, we propose a new method to reduce the communication cost.

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, $f(x) = 0$. Due to space limitations, we provide details of the method to extend DPF for additive shares and its correctness in the full version [14].

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 15 and Table 16).

[5]Note that 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.

7 EXPERIMENTAL RESULTS

This section discusses the scalability of S^2, investigates the impact of different parameters on S^2, and compares S^2 against other systems. 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 ⟨SupplyKey (SK), PartKey (PK), LineNumber (LN), OrderKey (OK)⟩ 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. **Code:** is written in Java and contains more than 9K lines. **Time:** is calculated by taking an average of 10 runs of programs, shown in seconds (s).

7.1 S^2 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 S^2 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.

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 14 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.

Exp 2: Query execution performance. To evaluate the query performance, we run S^2 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 15 and Table 16 show time for each operation at different entities.

Maximum computation time at servers. Recall that S^2 partitions a selection query into a search and fetch query. In round one for searching the qualified row-ids, S^2 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

Table 14: Exp1: Share generation time & average size of tables.

Rows	Time for share creation & 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

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, 45, 72].

Maximum computation time at a client. Computation time for the client for any operation is significantly less than 1s (0.067s for 1M rows and 0.210s for 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.

Table 15: Exp 2: Time (s) breakdown on 1M rows via 1 thread.

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

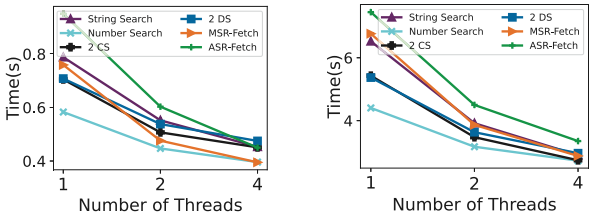
Entity	String search	Number search	Conjunctive search	Disjunctive search	Row Fetch – MSR	Row Fetch – ASR
Client	0.208	0.205	0.210	0.201	0.029	0.070
Server	6.315	4.030	5.201	5.163	6.694	8.478
Total	**6.523**	**4.235**	**5.411**	**5.364**	**6.723**	**8.548**

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.

Exp 3: Impact of parallelism. S^2 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 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, S^2 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 S^2 using 4-threaded implementation of S^2, 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, as computing fingerprints over more values. The computation time



(a) 1M Rows. **(b) 10M Rows.**

Figure 3: Exp 3: S^2 performance on multi-threaded implementation at AWS. Time in seconds.

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 ⟨OK,SK⟩ and ⟨PK,LN⟩; thus, $\mathcal{S}_c$ and client also work on more data to obtain the final answer. Note that in 4DS, servers cannot multiply all four column values; otherwise, 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.

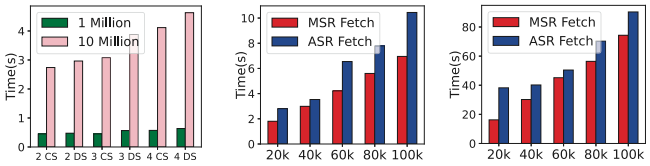
Exp 5: Data size and the impact of communication. In our approach, servers/combiner send data to a client to answer a query.

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 S^2. We considered three different speeds of data transfer: slow (50MB/s), medium (100MB/s), and fast (1GB/s). 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 S^2.

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



(a) Exp 4a: CS/DS. (b) Exp 4b: 1M Rows. (c) Exp 4b: 10M Rows.

Figure 4: Exp 4: Impact of different parameters.

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 burden on the client when a client is far from servers.

7.2 S^2 vs Other Systems

S^2 is information-theoretically secure; thus, we compare S^2 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 S^2 against those systems, Table 1 provides experimental results given in the respected papers. Below, we compare S^2 against the following additive sharing-based systems: Jana [19], Waldo [33], and Ciphercore [15]. Table 3 shows such results. Important to note that ***the existing secret-sharing systems do not support the large data and take more time compared to*** S^2.

Download methods. We compare S^2 against two download strategies, as mentioned in §1. Overall, S^2 outperforms such methods.

Jana [19]: supports only selection queries over additive shares in a single round of communication between a client and a server, while requiring servers to communicate among themselves for query execution. Jana converts all non-desired rows 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 keyword over additive shares. Waldo does not allow knowing row-ids where the keyword exists and 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, and requires servers to communicate among themselves to execute queries. Ciphercore returns a vector containing 1 or 0 to the client, where 1 means rows containing the query keyword. Current version of Ciphercore does not support operations to fetch the desired row. 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 min for creating shares and executing a search query over 1M rows, while S^2 took at most 8s (7.2s to create shares and 0.788s for a search query).

S^{3}ORAM [45]: 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. Current code allows searching only *unique* random numbers and incurs high space overhead by storing twice the amount of input numbers. Current code does not allow importing any dataset. We provide experimental results (taken from the paper) of S^{3}ORAM in Table 1. Note that, such numbers are not on our data.

8 RELATED WORK

Secret-Sharing-based solutions. Additive [20] and multiplicative [69] are the two famous secret-sharing techniques. Such techniques perform addition over shares efficiently locally at servers, while the multiplication of shares requires communication among servers [18]. Sharemind [20], SPDZ [30, 31], Jana [19], Conclave [72], and Waldo [33] use additive shares. PDAS [71], Obscure [44], and [34, 74] use multiplicative shares. 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 discussed in §1. Table 1 compares such techniques. In contrast, S^2 offers highly efficient query execution using both additive and multiplicative shares, and also, prevents leakages from both access-patterns and volume. S^2 does not use a trusted party.

Information leakage via access-patterns. [24, 35, 42, 47, 51, 53, 55, 60, 61] discuss the impacts of revealing access-patterns on encrypted data. To overcome leakages from access-patterns, ORAM [40, 41, 63] and their improved version called PathORAM [70] were developed. Such solutions have asymptotic complexity of polylogarithmic in the index size. However, all such solutions have multiple problems, as mentioned in §1. S^{3}ORAM [45] provides ORAM-type index for secret-shares, but suffers from several problems. PIR, DPF, and FSS also hide access-patterns. S^2 provides two access-pattern hiding methods for row fetch: one is information-theoretically secure and another is based on DPF for additive shares.

Information leakage via volume. [66] showed that even when hiding access-patterns, an adversary can learn based only on volume. [17, 48, 65, 68] are recent volume-hiding techniques for only encrypted data. These techniques incur significant storage overhead (by storing ciphertext that is at least twice the actual data [17, 48, 65]) and show inefficient query execution.

9 CONCLUSION

We develop S^2 — efficient and scalable techniques for selection queries, based on both additive and multiplicative secret-sharing. S^2 does not reveal information from ciphertext and query execution via both access-patterns and volume/output-size, simultaneously. S^2 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.

ACKNOWLEDGEMENTS

We are thankful to the reviewers. Yin Li was funded by National Key Research and Development Program of China under Grant 2021YFB3101300. Sharad Mehrotra was partially funded by the research sponsored by DARPA under agreement number FA8750-16-2-0021 and NSF Grants No. 1952247, 2133391, 2032525, and 2008993. Nisha Panwar was partially funded by NSF Grants No. 2131538.

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

[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: https://www.factioninc.com/blog/hybrid-multi-cloud/multi-cloud-trends/.

[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://github.com/SecretDeB/S2-VLDB-2023.

[15] Ciphercore GitHub. Available at: https://github.com/ciphermodelabs/ciphercore.

[16] I. Ahmad et al. Coeus: A system for oblivious document ranking and retrieval. In *SOSP*, pages 672–690, 2021.

[17] G. Amjad, S. Patel, G. Persiano, K. Yeo, and M. Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *IACR Cryptol. ePrint Arch.*, page 765, 2021.

[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] R. Bost et al. Thwarting leakage abuse attacks against searchable encryption - A formal approach and applications to database padding. *IACR Cryptol. ePrint Arch.*, page 1060, 2017.

[22] E. Boyle et al. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

[23] M. Burkhart et al. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, 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*, volume 8134, 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] P. Gupta et al. Obscure: Information-theoretic oblivious and verifiable aggregation queries. *PVLDB*, 12(9):1030–1043, 2019.

[45] 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.

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

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

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

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

[50] J. Katz and Y. Lindell. *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 in untrusted clouds. In *NSDI*, pages 1031–1056, 2023.

[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, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[64] S. Oya et al. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, pages 127–142, 2021.

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

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

[67] A. Rajan et al. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS*, pages 49:1–49:4, 2018.

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

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

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

[71] B. Thompson et al. Privacy-preserving computation and verification of aggregate queries on outsourced databases. In *PETS*, volume 5672, pages 185–201, 2009.

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

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

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

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