Prism: Private Verifiable Set Computation over Multi-Owner Outsourced Databases*

Yin Li,[1] Dhrubajyoti Ghosh,[2] Peeyush Gupta,[2] Sharad Mehrotra,[2] Nisha Panwar,[3]
Shantanu Sharma[2]
[1]Dongguan University of Technology, China. [2]University of California, Irvine, USA. [3]Augusta University, USA.
Email: sharad@ics.uci.edu, shantanu.sharma@uci.edu

ABSTRACT

This paper proposes Prism, a secret sharing based approach to compute private set operations (*i.e.*, intersection and union), as well as aggregates over outsourced databases belonging to multiple owners. Prism enables data owners to pre-load the data onto non-colluding servers and exploits the additive and multiplicative properties of secret-shares to compute the above-listed operations in (at most) two rounds of communication between the servers (storing the secret-shares) and the querier, resulting in a very efficient implementation. Also, Prism does not require communication among the servers and supports result verification techniques for each operation to detect malicious adversaries. Experimental results show that Prism scales both in terms of the number of data owners and database sizes, to which prior approaches do not scale.

ACM Reference Format:
Yin Li,[1] Dhrubajyoti Ghosh,[2] Peeyush Gupta,[2] Sharad Mehrotra,[2] Nisha Panwar,[3], Shantanu Sharma[2]. 2021. Prism: Private Verifiable Set Computation over Multi-Owner Outsourced Databases. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452839

1 INTRODUCTION

With the advent of cloud computing, database-as-a-service (DaS) [29] has gained significant attention. Traditionally, the DaS problem focused on a single database (DB) owner, submitting suitably encrypted data to the cloud over which DB owner (or one of its clients) can execute queries. A more general use-case is one in which there are multiple datasets, each owned by a different owner. Data owners do not trust each other, but wish to execute queries over common attributes of the dataset. The query execution must not reveal the content of the database belonging to one DB owner to others, except for the leakage that may occur from the answer to the query. The most common form of such queries is the *private set intersection* (PSI) [23]. An example use-case of PSI include *syndromic surveillance*, wherein organizations, such as pharmacies and hospitals share information (*e.g.*, a sudden increase in sales of specific drugs such as analgesics or anti-allergy medicine, telehealth calls, and school absenteeism requests) to enable early detection of community-wide outbreaks of diseases. PSI is also a building block for performing joins across private databases — it essentially corresponds to a semi-join operation on the join attribute [38].

Private set computations over datasets owned by different DB owners/organizations can, in general, be implemented using secure multiparty computation (SMC) [26, 44, 57], a well-known cryptographic technique that has been prevalent for more than three decades. SMC allows DB owners to securely execute any function over their datasets without revealing their data to other DB owners. However, SMC can be very slow, often by order of magnitude [41]. Consequently, techniques that can more efficiently compute private set operations have been developed; particularly, in the context of PSI and *private set union* (PSU) [19, 40]. PSU refers to privately computing the union of all databases. Approaches using homomorphic encryption [15], polynomial evaluation [23], garbled-circuit techniques [32], Bloom-filter [47], and oblivious transfer [49, 50] have been proposed to implement private set operations.

Recent work on private set operations has also explored performing aggregation on the result of PSI operations. For instance, [34] studied the problem of private set intersection sum (PSI Sum), motivated by the internet advertising use-case, where a party maintains information about which customer clicked on specific advertisements during their web session, while another has a list of transactions about items listed in the advertisements that resulted in a purchase by the customers. Both parties might wish to securely learn the total sales that attributed due to customers clicking on advertisements, while neither would like their data to be revealed to the other for reasons including fair/competitive business strategies.

Existing approaches on private set computation (including recent work on aggregation) are limited in several ways:

- Work on PSI or PSU has largely focused on the case of two DB owners, with some exceptions that address more than two DB owners scenarios, *e.g.*, [16, 23, 31, 33, 40, 41, 58]. There are several interesting use-cases, where one may wish to compute PSI over multiple datasets. For instance, in the syndromic surveillance example listed above, one may wish to compute intersection amongst several independently owned databases. Generalizing existing two-party PSI or PSU approaches to the case of multiple DB owners results in significant overhead [41]. For instance, [3], which is designed for two DB owners, incurs $(nm)^2$ communication cost, when extended to $m > 2$ DB owners, where n is the dataset size.

*This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is partially supported by NSF grants 1527536 and 1545071. Yin Li is supported by the National Natural Science Foundation of China (Grant no. 61972090, 61972089, 61601396).

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452839

	Name	Age	Disease	Cost
τ_1	John	4	Cancer	100
τ_2	Adam	6	Cancer	200
τ_3	Mike	2	Heart	300

Table 1: Hospital 1.

	Name	Age	Disease	Cost
ν_1	John	8	Cancer	100
ν_2	Adam	5	Fever	70
ν_3	Bob	4	Fever	50

Table 2: Hospital 2.

	Name	Age	Disease	Cost
ρ_1	Carl	8	Cancer	300
ρ_2	John	4	Cancer	700
ρ_3	Lisa	5	Heart	500

Table 3: Hospital 3.

Note: τ_i, ν_i, and ρ_i denote the i^{th} tuples of tables.

- Techniques to privately compute aggregation over set operations have not been studied systematically. In database literature, aggregation functions [46] are typically classified as: *summary* aggregations (*e.g.*, count, sum, and average) or *exemplary* aggregations (*e.g.*, minimum, maximum, and median). Existing literature has only considered the problem of PSI Sum [34] and cardinality determination, *i.e.*, the size of the intersection/union [19, 22]. Techniques for exemplary aggregations (and even for summary aggregations) that may compute over multiple attributes have not been explored.
- Many of the existing solutions do not deal with a large amount of data, due to either inefficient cryptographic techniques or multiple communication rounds amongst DB owners. For instance, recent work [41, 42, 58] dealt with data that is limited to sets of size less than or equal to ≈1M in size.

This paper introduces Prism — a novel approach for computing collaboratively over multiple databases. Prism is designed for both PSI and PSU, and supports both summary, as well as, exemplar aggregations. Unlike existing SMC techniques (wherein DB owners compute operations privately through a sequence of communication rounds), in Prism, DB owners outsource their data in secret-shared form to multiple ***non-communicating public servers***. As will become clear, Prism exploits the homomorphic nature of secret-shares to enable servers to compute private set operations independently (to a large degree). These results are then returned to DB owners to compute the final results. In Prism, any operator requires at most two communication rounds between DB owners and servers, where the first round finds tuples that are in the intersection or union of the set, and the second round computes the aggregation function over the objects in the intersection/union.

By using public servers for computation over secret-shared data, Prism achieves the identical security guarantees as existing SMC systems (*e.g.*, Sharemind [8], Jana [5], and Conclave [54]). The key advantage of Prism is that by outsourcing data in secret shared form and exploiting homomorphic properties, Prism does not require communication among server before/during/after the computation, which allows Prism to perform efficiently even for large data sizes and for a large number of DB owners (as we will show in experiment section). Since Prism uses the public servers, which may act maliciously, Prism supports oblivious result verification methods.

Advantages of Prism. In summary, Prism offers the following benefits: (*i*) *Information-theoretical security*: It achieves information-theoretical security at the servers and prevents them to learn anything from input/output/access-patterns/output-size. (*ii*) *No communication among servers*: It does not require any communication among servers, unlike SMC-based solutions. (*iii*) *No trusted entity*: It does not require any trusted entity that performs the computation on the cleartext data, unlike the recent SMC system Conclave [54]. (*iv*) *Several DB owners and large-sized dataset*: It deals with several DB owners having a large-size dataset.

Full version [1]. provides result verification methods for different aggregation approaches, correctness, and information leakage discussions.

2 PRIVATE SET OPERATIONS

We, first, define the set of operations supported by Prism. Let $DB_1, \ldots, DB_m$ ($m > 2$) be independent DBs owned by m DB owners $\mathcal{DB}_1, \ldots, \mathcal{DB}_m$. We assume, each DB owner is (partially) aware of the schema of data stored at other DB owners. Particularly, DB owners have knowledge of attribute(s) of the data stored at other DB owners on which the set-based operations (intersection/union) can be performed. Also, DB owners know about the attributes on which aggregation functions be supported. This assumption is needed to ensure that PSI/PSU and aggregation queries are well defined. However, the schema of data at different databases may be different.

Now, we define the private set operations supported by Prism formally and their corresponding privacy requirements (corresponding SQL statements are shown in Table 4). To do so, (and in the rest of the paper), we use the example tables shown in Tables 1, 2, and 3 that are owned by three different DB owners (in our case, hospitals).

(1) ***Private set intersection (PSI)* (§5).** PSI finds the common values among m DB owners for a specific attribute A_c, *i.e.*, $DB_1.A_c \cap \ldots \cap DB_m.A_c$. For example, PSI over disease column of Tables 1, 2, and 3 returns {Cancer} as a common disease treated by all hospitals. Note that *a hospital computing PSI on disease should not gain any information about other possible disease values (except for the result of the PSI) associated with other hospitals.*

(2) ***Private set union (PSU)* (§7).** PSU finds the union of values among m DB owners for a specific attribute A_c, *i.e.*, $DB_1.A_c \cup \ldots \cup DB_m.A_c$. E.g., PSU over disease column returns {Cancer, Fever, Heart} as diseases treated by all hospitals. *A hospital computing PSU over other hospitals must not gain information about the specific diseases treated by others, or how many hospitals treat which disease.*

(3) ***Aggregation over private set operators (§6.)*** Aggregation ${}_{A_c}\mathcal{G}_{\theta}(A_x)$ computes an aggregation function θ on attribute A_x ($A_c \neq A_x$) for the groups corresponding to the output of set-based operations (PSI/PSU) on attribute A_c. E.g., the aggregation function *sum* on cost attribute corresponding to PSI over disease attribute (*i.e.*, ${}_{\text{disease}}\mathcal{G}_{sum}(\text{cost})$) returns a tuple {Cancer,1400}. The same aggregation function over PSU will return {⟨Cancer,1400⟩, ⟨Fever,120 ⟩, ⟨ Heart,800⟩}. Likewise, the output of aggregation ${}_{\text{disease}}\mathcal{G}_{max}(\text{age})$ over PSI would return {Cancer,8}, while the same over PSU would return {⟨Cancer,8⟩, ⟨Fever,5⟩, ⟨Heart,5⟩}. Note that the count operation does not require specifying an aggregation attribute A_x and can be computed over the attribute(s) associated with PSI/PSU. E.g., count over PSI (PSU) on disease column will return 1 (3), respectively. From the perspective of privacy requirement, in the case of PSI on disease column, a hospital executing an aggregation query (maximum of age or sum of cost) should only gain information about the answer, *i.e.*, *elements in the PSI and the corresponding aggregate value.* It should not gain information about other diseases that are not in the intersection. Likewise, for PSU, the hospital will gain information about *all elements in the union and their corresponding aggregate values, but will not gain any specific information about which database contains which disease values, or the number of databases with a specific disease.*

PSI	SELECT A_c FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m
PSU	SELECT A_c FROM db_1 UNION ... UNION SELECT A_c FROM db_m
PSI count	SELECT COUNT(A_c) FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m

PSI θ
$\theta \in$ (AVG, SUM, MAX, MIN, Median)
	CREATE VIEW $CommonA_c$ as SELECT A_c FROM db_1 INTERSECT ... INTERSECT SELECT A_c FROM db_m
	SELECT A_c, $\theta(A_x)$ FROM (SELECT A_x, A_c FROM db_1, $CommonA_c$ WHERE $db_1.A_c$ = $CommonA_c.A_c$ UNION ALL ... UNION ALL
	SELECT A_x, A_c FROM db_m, $CommonA_c$ WHERE $db_m.A_c$ = $CommonA_c.A_c$) as inner_relation Group By A_c

Table 4: SQL syntax of operations supported by Prism.

3 PRELIMINARY

This section presents the cryptographic concepts that serve as building blocks for Prism, an overview of Prism, and security properties.

3.1 Building Blocks

Prism is based on additive secret-sharing (SS), Shamir's secret-sharing (SSS), cyclic group, and pseudorandom number generator. We provide an overview of these techniques, below.

Additive Secret-Sharing (SS). Additive SS is the simplest type of the SS. Let δ be a prime number. Let $\mathbb{G}_\delta$ be an Abelian group under modulo addition δ operation. All additive shares are defined over $\mathbb{G}_\delta$. In particular, the DB owner creates c shares $A(s)^1, A(s)^2, \ldots, A(s)^c$ over $\mathbb{G}_\delta$ of a secret, say s, such that $s = A(s)^1 + A(s)^2 + \ldots + A(s)^c$. The DB owner sends share $A(s)^i$ to the i^{th} server (belonging to a set of c non-communicating servers). These servers cannot know the secret s until they collect all c shares. To reconstruct s, the DB owner collects all the shares and adds them. Additive SS allows *additive homomorphism*. Thus, servers holding shares of different secrets can locally compute the sum of those shares. Let $A(x)^i$ and $A(y)^i$ be additive shares of two secrets x and y, respectively, at a server i, then the server i can compute $A(x)^i + A(y)^i$ that enable DB owner to know the result of $x + y$. The precondition of *additive homomorphism is that the sum of shares should be less than* δ.

Example. Let $\mathbb{G}_5 = \{0, 1, 2, 3, 4\}$ be an Abelian group under the addition modulo 5. Let 4 be a secret. A DB owner may create two shares: 3 and 1 (since $4 = (3 + 1) \bmod 5$).

Shamir's Secret-Sharing (SSS) [52]. Let s be a secret. A DB owner randomly selects a polynomial of degree c' with c' random coefficients, *i.e.*, $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{c'}x^{c'}$, where $f(x) \in \mathbb{F}_p[x]$, p is a prime number, $\mathbb{F}_p$ is a finite field of order p, $a_0 = s$, and $a_i \in \mathbb{N}$ ($1 \leq i \leq c'$). The DB owner distributes s into c shares by computing $f(x)$ ($x = 1, \ldots, c$) and sends an i^{th} share to an i^{th} server (belonging to a set of c non-colluding servers). The secret can be reconstructed using any $c' + 1$ shares using Lagrange interpolation [18]. SSS allows *additive homomorphism*, *i.e.*, if $S(x)^i$ and $S(y)^i$ are SSS of two secrets x and y, respectively, at a server i, then the server i can compute $S(x)^i + S(y)^i$, which will result in $x + y$ at DB owner.

Cyclic group under modulo multiplication. Let η be a prime number. A group $\mathbb{G}$ is called a cyclic group, if there exists an element $g \in \mathbb{G}$, such that all $x \in \mathbb{G}$ can be derived as $x = (g^i)$ (where i in an integer number $\mathbb{Z}$) under modulo multiplicative η operation. The element g is called a generator of the cyclic group. The number of elements in $\mathbb{G}$ is called the *order* of $\mathbb{G}$. Based on each element x of a cyclic group, we can form a cyclic subgroup by executing $x^i \bmod \eta$.

Example. $g = 2$ is a generator of a cyclic group under multiplication modulo $\eta = 11$ for the group: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Note that the group elements are derived by $2^i \bmod 11$. By taking the element 5 of this cyclic group, we form the following cyclic subgroup $\{1, 3, 4, 5, 9\}$, under multiplication modulo $\eta = 11$, by $5^i \bmod 11$.

Permutation function $\mathcal{PF}$. Let A be a set. A permutation function $\mathcal{PF}$ is a bijective function that maps a permutation of A to another permutation of A, *i.e.*, $\mathcal{PF} : A \rightarrow A$.

Pseudorandom number generator $\mathcal{PRG}$: is a deterministic and efficient algorithm that generates a pseudorandom number sequence based on an input seed [7, 25].

3.2 Entities and Trust Assumption

Prism assumes the following four entities:

(1) The m **database (DB) owners** (or users), who wish to execute computation on their joint datasets. We assume that each DB owner is trusted and does not act maliciously.

(2) A set of $c \geq 2$ ***servers*** that store the secret-shared data outsourced by DB owners and execute the requested computation from authenticated DB owners. Data transmission between a DB owner and a server takes place in encrypted form or using anonymous routing [27] to prevent the locations of all servers from an adversary.

We assume that servers do not maliciously communicate (*i.e.*, non-communicating servers) with each other in violation of Prism protocols. Unlike other MPC mechanisms [8], (as will be clear soon), Prism protocols do not require the servers to communicate before/during/after the execution of the query. The security of secret-sharing techniques requires that out of the c servers, no more than $c' < c$ communicate maliciously or collude with each other, where c' is a minority of servers (*i.e.*, less than half of c). Thus, we assume that a majority of servers do not collude and communicate with each other, and hence, a legal secret value cannot be generated/inserted/updated/deleted at the majority of the servers. Also, note that the collusion of servers in violation of the protocol is a general requirement for secret-sharing based protocols, and a similar assumption is made by many prior work [8, 17, 52, 56]. This assumption is based on factors such as economic incentivization (violation is against their economic interest), law (illegal to collude), and jurisdictional boundaries. Such servers can be selected on different clouds, which make the assumption more realistic.

For the purpose of simplicity, we assume, none of the servers colludes with each other, *i.e.*, they do not communicate directly. Thus, to reconstruct the original secret value from the shares, *two additive shares* suffice. In the case of PSI sum (as in §6.1), we need to multiply two shares, each of degree one, and that increases the degree of the polynomial to two. To reconstruct the secret value of degree two, we need at least three multiplicative shares.

While we assume that servers do not collude, we consider two types of adversarial models for servers in the context of the computation that they perform: (*i*) ***Honest-but-curious*** (HBC) servers: correctly compute the assigned task without tampering with data or hiding answers. It may exploit side information (*e.g.*, the internal state of the server, query execution, background knowledge, and output size) to gain information about stored data, computation, or results. HBC adversarial model is considered widely in many cryptographic algorithms [13, 29, 55]. (*ii*) ***Malicious*** adversarial servers: can delete/insert tuples from the relation, and hence, is a stronger adversarial model than HBC.

(3) An ***initiator* or *oracle***, who knows m DB owners and servers. Before data outsourcing by DB owners, the initiator informs the identity of servers to DB owners and vice versa. Also, the initiator informs the desired parameters (*e.g.*, a hash function, parameters related to Abelian and cyclic groups, $\mathcal{PF}$, and $\mathcal{RRG}$) to servers and DB owners. The initiator is an entity trusted by all other entities and plays a role similar to the trusted certificate authority in the public-key infrastructure. The initiator never knows the data/results, since it does not store any data, or data/results are not provided to servers via the initiator. The role of the initiator has also been considered in existing PSI work [51, 59].

(4) An **announcer** $\mathcal{S}_a$ who participates only in maximum, minimum, and median queries to announce the results. $\mathcal{S}_a$ communicates (not maliciously) with servers and initiator (and not with DB owners).

3.3 Prism Overview

Let us first understand the working of Prism at the high-level. Prism contains four phases (see Figure 1), as follows:

Phase 0: *Initialization.* The initiator sends desired parameters (see details in §4) related to additive SS, SSS, cyclic group, $\mathcal{PF}$, and $\mathcal{PRG}$ to all entities and informs them about the identity of others from/to whom they will receive/send the data.

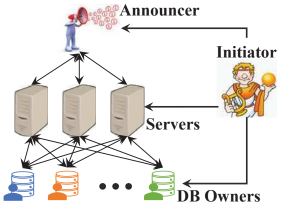


Figure 1: Prism model.

Phase 1: *Data Outsourcing by DB owners.* DB owners create additive SS or SSS of their data, by following methods given in §5 for PSI and PSU, §6.1 for PSI/PSU-sum, and §6.3 for PSI/PSU-max/min. Then, they outsource their secret-shared data to non-communicating servers. Note that in our explanations, we will write the data outsourcing method along with query execution.

Phase 2: *Query Generation by the DB owner.* A DB owner who wishes to execute SMC over datasets of different DB owners, sends the query to the servers. For generating secret-shared queries for PSI, PSU, count, sum, maximum, and for their verification, the DB owner follows the method given in §5, §6.

Phase 3: *Query Processing.* Servers process an input query and respective verification method in an oblivious manner. Neither the query nor the results satisfying the query/verification are revealed to the server. Finally, servers transfer their outputs to DB owners.

Phase 4: *Final processing at the DB owners.* The DB owner either adds the additive shares or performs Lagrange interpolation on SSS to obtain the answer to the query.

3.4 Security Property

As mentioned in the adversarial setting in §3.2, an adversarial server wishes to learn the (entire/partial) input and output data, while a DB owner may wish to know the data of other DB owners. Thus, a secure algorithm must prevent an adversary to learn the data (*i*) from the ciphertext representation of the data, (*ii*) from query execution due to access-patterns (*i.e.*, the adversary can learn the physical locations of tuples that are accessed to answer the query), and (*iii*) from the size of the output (*i.e.*, the adversary can learn the number of tuples satisfy the query). The attacks on a dataset based on access-patterns and output-size are discussed in [14, 35]. In order to prevent these attacks, our security properties are identical to the standard security definition as in [12, 13, 24]. An algorithm is *privacy-preserving* if it maintains DB owners' privacy, data/computation privacy from the servers, and performs identical operations regardless of the inputs.

Privacy from servers requires that datasets of DB owners must be hidden from servers, before/during/after any computation. In PSI/ PSU, servers must not know whether a value is common or not, the number of DB owners having a particular value in the result set. In the case of aggregation operations, the output of aggregation over an attribute A_x corresponding to the attributes A_c involved in PSI or PSU should not be revealed to servers. Also, in the case of max/median/min query, servers must not know the max/median/min value and the identity of the DB owner who possesses such values. Further, the protocol must ensure that the server's behavior in reading/sending the data must be identical for a particular type of query (*e.g.*, PSI or PSU), thereby the server should not learn anything from query execution (*i.e.*, hiding access-patterns and output-sizes).

DB owner privacy requires that the DB owners must not learn anything other than their datasets and the final output of the computation. For example, in PSI/PSU queries, DB owners must only learn the intersection/union set, and they must not learn the number of DB owners that does not contain a particular value in their datasets. Similarly, in the case of aggregation operations, DB owners must only learn the output of aggregation operation, not the individual values on which aggregation was performed.

Properties of verification. A verification method must be oblivious and find misbehavior of servers in computing a query. We follow the verification properties from [36] that the verification method cannot be refuted by the majority of the servers and should not leak any additional information.

4 ASSUMPTIONS & PARAMETERS

Different entities in Prism protocols are aware of the following parameters to execute the desired task:

Parameters known to the initiator. The initiator knows all parameters used in Prism and distributes them to different entities (only once) as they join in PRISM protocols. Note that the initiator can select these parameters (such as η, δ) to be large to support increasing DB owners over time without updating parameters. Thus, when new DB owners join, the initiator simply needs to inform DB owners/servers about the increase in the number of DB owners in the system, but does not need to change *all* parameters.

Additionally, the initiator does the following: (*i*) Selects a polynomial ($\mathcal{F}(x) = a_{m+1}x^{m+1} + a_m x^m + \ldots + a_1 x + a_0$, where $a_i > 0$) of degree more than m, where m is the number of DB owners, and sends the polynomial to all DB owners. This polynomial will be used during the maximum computation. Importantly, this polynomial $\mathcal{F}(x)$ generates values at different DB owners in an order-preserving manner, as will be clear in §6.3, and the degree of the polynomial must be more than m to prevent an entity, who has m different values generated using this polynomial, to reconstruct the secret value (a condition similar to SSS); and beyond $m + 1$, the degree of the polynomial does not impact the security, in this case. (*ii*) Generates a permutation function $\mathcal{PF}_i$, and produces four different permutation functions that satisfy Equation 1:

$$\mathcal{PF}_{s1} \odot \mathcal{PF}_{db1} = \mathcal{PF}_{s2} \odot \mathcal{PF}_{db2} = \mathcal{PF}_i \quad (1)$$

Symbol $\odot$ shows composition of permutations, and these functions can be selected over a permutation group. The initiator provides $\mathcal{PF}_{s1}, \mathcal{PF}_{s2}$ to all servers and $\mathcal{PF}_{db1}, \mathcal{PF}_{db2}$ to all DB owners.

Parameters known to announcer. Announcer $\mathcal{S}_a$ knows δ, a prime number used in modulo addition for an Abelian group (§3.1).

Parameters known to DB owners. All DB owners know the following parameters: (*i*) m, *i.e.*, the number of DB owners. (*ii*) $\delta > m$, (*iii*) η, where η is a prime number used to define modular multiplication for a cyclic group (§3.1). Note that DB owners do not know the generator g of the cyclic group. (*iv*) A common hash function. (*v*) The domain of the attribute A_c on which they want to execute PSI/PSU. Note that knowing the domain of the attribute A_c does not reveal that which of the DB owner has a value of the domain. (Such an assumption is also considered in prior work [32].) (*vi*) Two permutation functions $\mathcal{PF}_{db1}$ and $\mathcal{PF}_{db2}$. (*vii*) The polynomial $\mathcal{F}(x)$ given by the initiator. (*viii*) A permutation function $\mathcal{PF}$, and the same permutation function will also known to servers.

PSI, PSU, sum, average, count algorithms are based on the assumptions 1-5. PSI verification, sum verification, count, and count verification are based on the assumptions 1-6. Maximum, its verification, and median algorithms are based on assumptions 1-8.

We assume, any DB owner or the initiator provides additive shares of m to servers for executing PSI, and the DB owners have only positive integers to compute the max. Since the current PSI maximum method uses modular operations (as will be clear in §6.3), we cannot handle floating-point values *directly*. Nonetheless, we can find the maximum for a large class of practical situations, where the precision of decimal is limited, say $k > 0$ digits by simply multiplying each number by 10^k and using the current PSI maximum algorithm. E.g., we can find the maximum over $\{0.5, 8.2, 8.02\}$ by computing the maximum over $\{50, 820, 802\}$. Designing a more general solution that does not require limited precision is non-trivial.

Parameters known to servers. Servers know following parameters: (*i*) m, $\delta > m$, the generator g of the cyclic (sub)group of order δ and $\eta' = \alpha \times \eta$ and $\alpha > 1$. Based on the group theory, $\eta - 1$ should be divisible by δ. Note, servers do not know η. (*ii*) A permutation function $\mathcal{PF}$, and recall that the same permutation function is also known to DB owners. (*iii*) Two permutation functions $\mathcal{PF}_{s1}$ and $\mathcal{PF}_{s2}$. (*iv*) A common pseudo-random number generator $\mathcal{PRG}$ that generates random numbers between 1 and $\delta - 1$; $\mathcal{PRG}$ is unknown to DB owners. PSI, sum, and average are based on the assumptions 1. Maximum, its verification, and median are based on the assumptions 1,2. Count and its verification are based on the assumptions 1,3. PSU algorithm is based on the assumptions 1,4.

5 PRIVATE SET INTERSECTION QUERY

This section, first, develops a method for finding PSI among $m > 2$ different DB owners on an attribute A_c (which is assumed to exist at all DB owners, §5.1) and presents a result verification method (§5.2). Later in §6.6, we present a method to execute PSI over multiple attributes and a method to reduce the communication cost of PSI.

5.1 PSI Query Execution

High-level idea. Each of $m > 2$ DB owners uses a publicly known hash function to map distinct values of A_c attribute in a table of at most $|\text{Dom}(A_c)|$ cells, where $|\text{Dom}(A_c)|$ is the size of the domain of A_c. Thus, if a value $a_j \in A_c$ exists at any DB owner, all DB owners must map a_j to an identical cell of the table. All values of the table are outsourced in the form of additive shares to *two non-communicating servers* $\mathcal{S}_\phi$, $\phi \in \{1, 2\}$, that *obliviously* find the common items/intersection and return shared output vector (of the same length as the length of the received shares from DB owners). Finally, each DB owner adds the results to know the final answer.

Construction. We create the following construction over elements of a group under addition and elements of a cyclic group under multiplication. We can select any cyclic group such that $\eta > m$.

$$(x + y) \bmod \delta = 0, (g^x \times g^y) \bmod \eta = 1 \quad (2)$$

Based on this construction, below, we explain PSI finding algorithm:

Step 1: DB owners. Each DB owner finds distinct values in an attribute (A_c, which exists at all DB owners, as per our assumption given in §4) and executes the hash function on each value a_i to create a table $\chi = \{x_1, x_2, \dots, x_b\}$ of length $b = |\text{Dom}(A_c)|$. The hash function maps the value $a_i \in A_c$ to one of the cells of χ, such that the cell of χ corresponding to the value a_i holds 1; otherwise 0.[1] It is important that each cell must contain only a single one corresponding to the unique value of the attribute A_c, and note that if a value $a_i \in A_c$ exists at any DB owner, then one corresponding to a_i is placed at an identical cell of χ at the DB owner. The table at $\mathcal{DB}_j$ is denoted by χ_j. Finally, $\mathcal{DB}_j$ creates additive secret-shares of each value of χ_j (*i.e.*, additive secret-shares of either one or zero) and outsources the ϕ^{th}, $\phi \in \{1, 2\}$, share to the server $\mathcal{S}_\phi$. We use the notation $A(x_i)_j^\phi$ to refer to ϕ^{th} additive share of an i^{th} element of χ_j of $\mathcal{DB}_j$. Recall that before the computation starts, the initiator informs the locations of servers to DB owners and vice versa (§3.2).

Step 2: Servers. Each server $\mathcal{S}_\phi$ ($\phi \in \{1, 2\}$) holds the ϕ^{th} additive share of the table χ (denoted by $A(\chi)_j^\phi$) of j^{th} $(1 \leq j \leq m)$ DB owners and executes Equation 3:

$$output_i^{\mathcal{S}_\phi} \leftarrow g^{((\oplus_{j=1}^{j=m} A(x_i)_j^\phi) \ominus A(m)^\phi)} \bmod \eta', (1 \leq i \leq b) \quad (3)$$

where $\oplus$ and $\ominus$ show the modular addition and modular subtraction operations, respectively. We used the symbols $\oplus$ and $\ominus$ to distinguish them from the normal addition and subtraction. Particularly, each server $\mathcal{S}_\phi$ performs the following operations: (*i*) modular addition (under δ) of the i^{th} additive secret-shares from all m DB owners, (*ii*) modular subtraction (under δ) of the result of the previous step from the additive share of m (*i.e.*, $A(m)^\phi$), (*iii*) exponentiation by g to the power the result of the previous step and modulo by η', and (*iv*) sends all the computed b results to the m DB owners.

Step 3: DB owners. From two servers, DB owners receive two vectors, each of length b, and perform modular multiplication (under η) of outputs $output_i^{\mathcal{S}_1}$ and $output_i^{\mathcal{S}_2}$, where $1 \leq i \leq b$, *i.e.*,

$$fop_i \leftarrow (output_i^{\mathcal{S}_1} \times output_i^{\mathcal{S}_2}) \bmod \eta \quad (4)$$

This step results in an output array of b elements, which may contain any value. However, if an i^{th} item of χ_j exists at all DB

[1] We can also add any positive random number except 1 in case of 0 to prevent revealing data distribution based on background knowledge; see [1] for details.

Value	Share 1	Share 2
1	4	-3
0	2	-2
1	3	-2

Table 5: $\mathcal{DB}_1$.

Value	Share 1	Share 2
1	3	-2
1	4	-3
0	3	-3

Table 6: $\mathcal{DB}_2$.

Value	Share 1	Share 2
1	2	-1
0	3	-3
1	4	-3

Table 7: $\mathcal{DB}_3$.

owners, then fop_i must be one, since $\mathcal{S}_\phi$ have added additive shares of m ones at the i^{th} element and subtracted from additive share of m that results in (g^0 mod η') mod $\eta = 1$ at DB owner. Please see the correctness argument below after the example.

Example 5.1. Assume three DB owners: $\mathcal{DB}_1$, $\mathcal{DB}_2$, and $\mathcal{DB}_3$; see Tables 1, 2, and 3. For answering a query to find the common disease that is treated by each hospital, DB owners create their tables χ as shown in the first column of Tables 5, 6, and 7. For example, in Table 6, $\langle 1, 1, 0\rangle$ corresponds to cancer, fever, and heart diseases, where 1 means that the disease is treated by the hospital. We select $\delta = 5$, $\eta = 11$, and $\eta' = 143$. Hence, the Abelian group under modulo addition contains $\{0, 1, 2, 3, 4\}$, and the cyclic (sub)group (with $g = 3$) under modulo multiplication contains $\{1, 3, 4, 5, 9\}$. Assume additive shares of $m = 3 = (1 + 2) \bmod 5$.

Step 1: DB Owners. DB owners generate additive shares as shown in the second and third columns of Tables 5, 6, and 7, and outsource all values of the second and third columns to $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively.

Step 2: Servers. The server $\mathcal{S}_1$ will return the three values 27, 27, 81, by executing the following computation, to all three DB owners:

$$3^{((((4+3+2) \bmod 5)-1) \bmod 5)} \bmod 143 = 27$$
$$3^{((((2+4+3) \bmod 5)-1) \bmod 5)} \bmod 143 = 27$$
$$3^{((((3+3+4) \bmod 5)-1) \bmod 5)} \bmod 143 = 81$$

The server $\mathcal{S}_2$ will return values 9, 1, and 1 to all three DB owners:

$$3^{((((-3-2-1) \bmod 5)-2) \bmod 5)} \bmod 143 = 9$$
$$3^{((((-2-3-3) \bmod 5)-2) \bmod 5)} \bmod 143 = 1$$
$$3^{((((-2-3-3) \bmod 5)-2) \bmod 5)} \bmod 143 = 1$$

Step 3: DB owners. The DB owner obtains a vector $\langle 1, 5, 4\rangle$, by executing the following computation (see below). From the vector $\langle 1, 5, 4\rangle$, DB owners learn that cancer is a common disease treated by all three hospitals. However, the DB owner does not learn anything more than this; note that in the output vector, the values 5 and 4 correspond to zero. For instance, $\mathcal{DB}_1$, *i.e.*, hospital 1, cannot learn whether fever and heart diseases are treated by hospital 2, 3, or not.

$$(27 \times 9) \bmod 11 = 1 \quad (27 \times 1) \bmod 11 = 5 \quad (81 \times 1) \bmod 11 = 4$$

Correctness. When we plug Equation 3 into Equation 4, we obtain:

$$fop_i = (g^{(\oplus_{j=1}^{j=m} A(x_i)_j^1) \ominus A(m)^1} \times g^{(\oplus_{j=1}^{j=m} A(x_i)_j^2) \ominus A(m)^2} \bmod \eta') \bmod \eta$$
$$= (g^{(\oplus_{j=1}^{j=m} (x_i)_j - m)} \bmod \eta') \bmod \eta$$

We utilize modular identity, *i.e.*, (x mod $\alpha\eta$) mod $\eta = x$ mod η; thus, $fop_i = g^{(\sum_{j=1}^{j=m} (x_i)_j - m)} \bmod \eta$. Only when $\sum_{j=1}^{j=m} (x_i)_j = m$, the result of above expression is one; otherwise, a nonzero number.

Information leakage discussion. We need to prevent information leakage at the server and at the DB owners.

(1) *Server perspective.* Servers only know the parameters $\langle g, \delta, \eta'\rangle$ and may utilize the relations between g and η to guess η from η'. However, it will not give any meaningful information to servers, since the DB owner sends the elements of χ in additive shared form, and since servers do not communicate with each other, they cannot obtain the cleartext values of χ. Also, an identical operation is executed on all shares of m DB owners. Hence, access-patterns are hidden from servers, preventing them to distinguish between any two values based on access-patterns. Also, the output of queries is in shared form and contains an identical number of bits as inputs. Thus, based on the output size, servers cannot know whether the value is common among DB owners or not.

(2) *DB owner perspective.* When all DB owners do not have one at the i^{th} position of χ, we need to inform DB owners that there is no common value and not to reveal that how many DB owners do not have one at the i^{th} position. Note that the DB owner can learn this information, if they know g and α, since based on these values, they can compute what the servers have computed. However, unawareness of g and α makes it impossible to guess the number of DB owners that do not have one at the i^{th} position of χ. We can formally prove it as follows:

Lemma. A DB owner cannot deduce how many other DB owners do not have one at the i^{th} position of χ without knowing g.

Proof. According to the precondition, g is a generator of a cyclic group of order δ, where δ is a prime number. Thus, $C = \{g^0, g, g^2, \ldots, g^{\delta-1}\}$ represents all items in the cyclic group. Assume that the output of Equation 4 is a number other than one, say β. Thus, we have $\beta = g^{x-m} \bmod \eta$, where x represents the number of one at the i^{th} position of χ_j, $1 \leq j \leq m$. When DB owners wish to know x, they must compute $\log_g \beta$. To solve it, they need to know g. Note that based on the characteristic of the cyclic group, there are less than $\delta - 1$ generators of C and co-prime to δ. Thus, $g^2, \ldots, g^{\delta-1}$ may also be generators of the cyclic group. However, DB owners cannot distinguish which generator is used by the servers. Thus, DB owners cannot deduce the value of x, except knowing that $x \in [0, m-1]$.[2]■

5.2 PSI Result Verification

A malicious adversary or a hardware/software bug may result in the following situations, during computing PSI: (*i*) skip processing the i^{th} additive shares of all DB owners, (*ii*) replacing the result of the i^{th} additive shares by the computed result for j^{th} share, (*iii*) injecting fake values, or (*iv*) falsifying the verification method. Thus, this section provides a method for verifying the result of PSI.

High-level idea. Let g be a generator of a cyclic group under modulo multiplicative η operation, and $\eta' = \alpha \times \eta$, $\alpha > 1$. Thus, (g^x mod η) × (g^{-x} mod η) = 1, and the idea of PSI verification lies in this equation. Recall, in PSI (§5.1), we used (g^x mod η), whose value 1 shows that the item exists at all DB owners. Now, we will use the term (g^{-x} mod η) for verification. Specifically, if the servers has performed their computations correctly, then Equation 5 must hold to be true:

$$((g^{(\oplus_{j=1}^{j=m} A(x_i)_j^\phi) - A(m)^\phi} \bmod \eta') \times (g^{\oplus_{j=1}^{j=m} \overline{A(x_i)_j^\phi}} \bmod \eta')) \bmod \eta = 1 \tag{5}$$

where m is the number of DB owners, x_j is either 1 or 0 (as described in §5.1), and $\overline{x_j}$ is the complement value of x_j. Below, we describe the steps executed at the servers and DB owners.

Step 1: DB owners. On distinct values of an attribute A_c of their relations, $\mathcal{DB}_j$ executes a hash function to create the table χ_j that

[2] Consider i^{th}, j^{th}, and k^{th} values of $\chi_1 = \{1, 0, 0\}$, $\chi_2 = \{0, 1, 0\}$, $\chi_3 = \{1, 1, 1\}$. Here, after Step 3, DB owners will learn three random numbers, such that the first two random numbers will be identical. Based on this, DB owner can only know that the sum of i^{th} and j^{th} position of χ is identical. However, it will not reveal how many positions have 0 or 1 at i^{th} or j^{th} positions.

$\overline{\text{Value}}$	Share 1	Share 2
0	2	-2
1	0	1
0	1	-1

Table 8: $\mathcal{DB}_1$.

$\overline{\text{Value}}$	Share 1	Share 2
0	2	-2
0	3	-3
1	4	-3

Table 9: $\mathcal{DB}_2$.

$\overline{\text{Value}}$	Share 1	Share 2
0	4	-4
1	1	0
0	1	-1

Table 10: $\mathcal{DB}_3$.

contains $b = |\text{Dom}(A_c)|$ values (either 0 or 1). Also, $\mathcal{DB}_j$ creates a table $\overline{\chi_j}$ containing b values, such that i^{th} value of $\overline{\chi_j}$ must be the complement of i^{th} value of χ_j. Then, $\mathcal{DB}_j$ permutes the values of $\overline{\chi_j}$ using a permutation function $\mathcal{PF}_{db1}$ (known to all DB owners *only*) and creates additive shares of each value of χ_j and $\overline{\chi_j}$, prior to outsourcing to servers. Reason of using $\mathcal{PF}_{db1}$ will be clear soon.

Step 2: Servers. Each server $\mathcal{S}_\phi$ holds the ϕ^{th} additive share of χ (denoted by $A(\chi)_j^\phi$) and $\overline{\chi}$ (denoted by $A(\overline{\chi})_j^\phi$) of j^{th} DB owner and executes the following operation:

$$output_i^{\mathcal{S}_\phi} \leftarrow g^{((\oplus_{j=1}^{j=m} A(x_i)_j^\phi) \ominus A(m)^\phi)} \bmod \eta', (1 \leq i \leq b) \quad (6)$$

$$Vout_i^{\mathcal{S}_\phi} \leftarrow g^{((\oplus_{j=1}^{j=m} A(\overline{x_i})_j^\phi))} \bmod \eta', (1 \leq i \leq b) \quad (7)$$

Equation 6 is identical to Equation 3 (in §5.1) and finds the common item at the server. In Equation 7, each server $\mathcal{S}_\phi$ performs following operations: (*i*) modular addition (under δ) of the i^{th} additive shares of $\overline{\chi}$ from m DB owners, (*ii*) exponentiation by g to the power the result of previous step, under modulo η'; and (*iii*) sends computed results $output^{\mathcal{S}_\phi}[]$ and $Vout^{\mathcal{S}_\phi}[]$ to DB owners.

Step 3: DB owners. From two servers, DB owners receive $output^{\mathcal{S}_\phi}[]$ and $Vout^{\mathcal{S}_\phi}[]$ (each of length b), permute back the values of $Vout^{\mathcal{S}_\phi}[]$ (using the reverse permutation function, since they used $\mathcal{PF}_{db1}$ on $\overline{\chi}$, which results in $Vout^{\mathcal{S}_\phi}[]$ at servers) to obtain $pvout^{\mathcal{S}_\phi}[]$, and execute the following:

$$r_1 \leftarrow output_i^{\mathcal{S}_1} \times output_i^{\mathcal{S}_2} \bmod \eta \quad (8)$$

$$r_2 \leftarrow pvout_i^{\mathcal{S}_1} \times pvout_i^{\mathcal{S}_2} \bmod \eta \quad (9)$$

$$r_1 \times r_2 \bmod \eta \mathrel{?} 1 \quad (10)$$

If the DB owner finds the output of $r_1 \times r_2$ equals one for all b values, it shows that the servers executed the computation correctly.

Example 5.2.1. We verify PSI results of Example 5.1.1. Suppose $\delta = 5$, $\eta = 11$, and $\eta' = 143$, as assumed in Example 5.1.1.

Step 1: DB owners. DB owners find the reverse of χ (as shown in the first column of Tables 8, 9, and 10) and generate additive shares; see the second and third columns of Tables 8, 9, and 10. Note that here for simplicity, we do not permute the values or shares.

Step 2: Servers. The server $\mathcal{S}_1$ will return the three values 27, 81, 3, by executing the following computation, to all three DB owners:

$$3^{((2+2+4) \bmod 5)} \bmod 143 = 27$$
$$3^{((0+3+1) \bmod 5)} \bmod 143 = 81$$
$$3^{((1+4+1) \bmod 5)} \bmod 143 = 3$$

$\mathcal{S}_2$ will return three values 7, 27, and 1 to all three DB owners:

$$3^{((-2-2-4) \bmod 5)} \bmod 143 = 9$$
$$3^{((1-3+0) \bmod 5)} \bmod 143 = 27$$
$$3^{((-1-3-1) \bmod 5)} \bmod 143 = 1$$

Step 3: DB owners. The DB owner obtains a vector $\langle 1, 9, 8 \rangle$, by executing the following computation:

$(27 \times 9) \bmod 11 = 1 \quad (81 \times 27) \bmod 11 = 9 \quad (3 \times 1) \bmod 11 = 3$

Now, the DB owner executes the following to verify PSI results: $1 \times 1 \bmod 11 = 1$, $5 \times 9 \bmod 11 = 1$, and $4 \times 3 \bmod 11 = 1$, where 1, 5, 4 are final outputs at DB owner in Example 5.1.1. The output 1 indicates that servers executed the computation correctly. ■

Correctness. First, we need to argue that the processing at servers works correctly. Assume that the DB owner does not implement $\mathcal{PF}_{db1}$ on elements of $\overline{\chi}$, and computation at servers is executed in cleartext. Thus, on the values of χ, servers add i^{th} value of each $\chi_j = \{x_i\}$ ($1 \leq j \leq m$, $1 \leq i \leq b$) and subtract the results from m. It will result in a number, say $a \in \{-m+1, 0\}$. On the other hand, servers add i^{th} values $\overline{\chi}_j$, and it will result in a number, say $b \in \{0, m\}$, *i.e.*, the number of ones at DB owners at the i^{th} position of χ. To hide the value of a and b from servers, they execute operations on additive shares of χ and $\overline{\chi}$, and take a modulus exponent (*i.e.*, $r_1 \leftarrow g^a$ and $r_2 \leftarrow g^b$) to hide a and b from DB owners. Since $a = -b$ or $a = b = 0$, $r_1 \times r_2 \bmod \eta = 1$, and this shows that the server executed the correct operation.

Now, we show why the verification method will detect any abnormal computation executed by servers. Note that servers may skip processing all/some values of χ and $\overline{\chi}$. For example, servers may process only $x_1 \in \chi$, $\overline{x_1} \in \overline{\chi}$, and send the results corresponding to $x_1, \overline{x_1}$ as the results of all remaining $b - 1$ values. Such a malicious operation of servers will provide legal proof (*i.e.*, $r_1 \times r_2 \bmod \eta = 1$) at DB owners that servers executed the computation correctly, (since values of $\overline{\chi}$ was not permuted). Thus, we used permutation over the values of $\overline{\chi}$ and/or additive shares of $\overline{\chi}$. Now, to break the verification method and to produce $r_1 \times r_2 \bmod \eta = 1$ for an i^{th} value of χ, servers need to find the correct value in $\overline{\chi}$ corresponding to an i^{th} value of χ (among the randomly permuted shares). Hence, the removal of any results from the output will be detected.

Now, we show that the verification method can detect fake data insertion by servers. For a server $\mathcal{S}_1$ to successfully inject a fake tuple (*i.e.*, undetected during verification), it should know the correct position of some element in both $A(\chi)_j^1$ and $A(\overline{\chi})_j^1$. Since $A(\overline{\chi})_j^1$ is a permuted vector of size $b = |\text{Dom}(A_c)|$, the probability of finding the correct element in $A(\overline{\chi})_j^1$ corresponding to an element of $A(\chi)_j^1$ will be $1/b^2$. E.g., in our experiments, the domain size is 5M (or 20M) values, making the above probability infinitesimal ($< 10^{-13}$).[3]

Additional security. We implemented $\mathcal{PF}_{db1}$ on the elements of $\overline{\chi}$. We can, further, permute additive shares of both χ and $\overline{\chi}$ using different permutation functions, to make it impossible for both servers to find the position of a value in $A(\chi)_j^\phi$ and $A(\overline{\chi})_j^\phi$, $\phi \in \{1, 2\}$. Thus, servers cannot break the verification method, and any malicious activities will be detected by DB owners.

Information leakages discussion. The verification method will not reveal any non-desired information to servers/DB owners, and arguments follow the similar way as for PSI computation in §5.1.

6 AGGREGATION OPERATION OVER PSI

Prism supports both summary and exemplar aggregations. Below, we describe how Prism implements sum §6.1, average §6.2, maximum §6.3, median §6.4 and count operations §6.5. Also, in our discussion below, we consider set-based operation PSI on a single attribute A_c. §6.6 extends the discussions to support PSI over multiple attributes and over a large-size domain. *Correctness and information leakage discussion of the following methods with their verification approaches are given in the full version [1].*

[3] If the domain size is small, we can increase its size by adding fake values to bind the probability of adversary being able to inject fake data.

6.1 PSI Sum Query

A PSI sum query computes the sum of values over an attribute corresponding to common items in another attribute; see example given in §2. This section develops a method based on additive, as well as, multiplicative shares, where additive shares find common items over an attribute A_c and multiplicative shares (SSS) finds the sum of shares of an attribute A_x corresponding to the common items in A_c. This method contains the following steps:

Step 1: DB owners. $\mathcal{DB}_j$ creates their χ_j table over the distinct values of A_c attribute by following Step 1 of PSI (§5). Here, $\chi_j = \{\langle x_{i1}, x_{i2}\rangle\}$ ($1 \leq i \leq b$ and $b = |\mathrm{Dom}(A_c)|$), *i.e.*, the i^{th} cell of χ_j contains a pair of values, $\langle x_{i1}, x_{i2}\rangle$, where (*i*) $x_{i1} = 1$, if a value $a_i \in A_c$ is mapped to the i^{th} cell, otherwise, 0; and (*ii*) x_{i2} contains the sum of values of A_x attribute corresponding to a_i; otherwise, 0. $\mathcal{DB}_j$ creates additive shares of x_{i1} (denoted by $A(x_{i1})_j^{\phi}$, $\phi = \{1, 2\}$) and sends to servers $\mathcal{S}_1$ and $\mathcal{S}_2$. $\mathcal{DB}_j$ also creates SSS of x_{i2} (denoted by $S(x_{i2})^{\phi=\{1,2,3\}}$) and sends to servers $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$.

Step 2: Servers. Servers $\mathcal{S}_1$ and $\mathcal{S}_2$ find common items using additive shares by implementing Equation 3 and send all computed b results to all DB owners. Since the result is in additive shared form, it cannot be multiplied to SSS. Thus, servers send the output of PSI to *one of the DB owners selected randomly* and wait to receive multiplicative shares corresponding to common items. The reason of randomly selecting only one DB owner is just to reduce the communication overhead of sending/receiving additive/multiplicative shares, and it does not impact the security. Note that all DB owners can receive the PSI outputs and generate multiplicative shares.

Step 3: DB owners. On receiving b values, the DB owner finds the common items by executing Equation 4 and generates a vector of length b having 1 or 0 only, where 0 is obtained by replacing random values of *fop*. Finally, DB owner creates three SSS of each b values, denoted by $S(z_i)^{\phi}$, $\phi = \{1, 2, 3\}$, and sends to three servers.

Step 4: Servers. Servers $\mathcal{S}_\phi$, $\phi = \{1, 2, 3\}$, execute the following:

$$sum_i^{\mathcal{S}_\phi} \leftarrow \sum_{j=1}^{j=m} S(x_{i2})_j^{\phi} \times S(z_i)^{\phi}, 1 \leq i \leq b \qquad (11)$$

Each server multiplies $S(z_i)^{\phi}$ by $S(x_{i2})_j^{\phi}$ of each DB owner, adds the results, and sends them to all DB owners.

Step 5: DB owners. From three servers, DB owners receive three vectors, each of length b, and perform Lagrange interpolation on each i^{th} value of the three vectors to obtain the final sum of the value in A_x corresponding to the common items in A_c.

6.2 PSI Average Query

A PSI average query on cost column corresponding to the common disease in Tables 1-3 returns {Cancer, 280}. PSI average query works in a similar way as PSI sum query. In short, $\mathcal{DB}_j$ creates $\chi_j = \{\langle x_{i1}, x_{i2}, x_{i3}\rangle\}$ ($1 \leq i \leq b$, $b = |\mathrm{Dom}(A_c)|$), and x_{i1}, x_{i2} are identical to the values we created in Step 1 of PSI sum (§6.1). The new value x_{i3} contains the number of tuples at $\mathcal{DB}_j$ corresponding to x_{i1}. E.g., in case of Table 1, one of the values of χ_1 will be {⟨Cancer, 300, 2⟩}, *i.e.*, Table 1 has two tuples corresponding to Cancer and cost 300. All values x_{i3} are outsourced in multiplicative share form. Then, we follow Steps 2 and 3 of PSI sum. In Step 4, servers multiply the received i^{th} SSS values corresponding to the common value to x_{i2}, x_{i3} and add the values. Finally, in Step 5, DB owners interpolate vectors corresponding to all b values of x_{i2}, x_{i3} and find the average by dividing the values appropriately.

6.3 PSI Maximum Query

This section develops a method for finding the maximum value in an attribute A_x corresponding to the common values in A_c attribute; refer to §2 for PSI maximum example. Here, our objective is to prevent the adversarial server from learning: (*i*) the actual maximum values outsourced by each DB owner, (*ii*) what is the maximum value among DB owners and which DB owners have the maximum value. We allow all the DB owners to know the maximum value and/or the identity of the DB owner(s) having the maximum value. We use pink color to highlight the part that is used to reveal the identity of DB owners having maximum to distinguish which part of the algorithm can be avoided based on the security requirements.

In this method, ***each DB owner uses polynomial $\mathcal{F}(x)$*** given by the initiator (see §4 to find how we created $\mathcal{F}(x)$). We use $\mathcal{F}(x)$ to generate values at different DB owners in an order-preserving manner by executing the following Step 3 and Equation 12.

The method contains at most three rounds, where the first round finds the common values in an attribute A_c by using Steps 1-3, the second round finds the maximum value in an attribute A_x corresponding to common items in A_c using Steps 4-5a, the last round finds DB owners who have the maximum value using Steps 5b-7. Note that ***the third round is not always required***, if (*i*) we do not want to reveal identity of the DB owner having the maximum value, or (*ii*) values in A_x column across all DB owners are unique.

Step 1 at DB owner and Step 2 at servers. These two steps are identical to Step 1 and Step 2 of PSI query execution method (§5).

Step 3: DB owner. On the received outputs (of Step 2) from servers, DB owners find the common item in the attribute A_c, as in Step 3 of PSI query execution method (§5). Now, to find the maximum value in the attribute A_x corresponding to the common item in A_c, DB owners proceeds as follows:

For simplicity, we assume that there is only one common item, say y^{th} item. $\mathcal{DB}_i$ finds the maximum, say $\mathcal{M}_{iy}$, in the attribute A_x of their relation corresponding to the common item y. Note that since we assume only one common element, we refer to the maximum element $\mathcal{M}_{iy}$ by $\mathcal{M}_i$. $\mathcal{DB}_i$ executes Equation 12 to produce values at DB owners in an order-preserving manner:

$$v_i \leftarrow \mathcal{F}(\mathcal{M}_i) + r_i \qquad (12)$$

$\mathcal{DB}_i$ implements the polynomial $\mathcal{F}()$ on $\mathcal{M}_i$ and adds a random number r_i (selected in a range between 0 and $\mathcal{M}_i^m$), and it produces a value v_i. Finally, $\mathcal{DB}_i$ creates additive shares of v_i (denoted by $A(v)_i^{\phi}$) and sends them to servers $\mathcal{S}_\phi$, $\phi = \{1, 2\}$. Note that even if $k \geq 2$ DB owners have the same maximum value $\mathcal{M}_i$, by this step, the value v will be different at those DB owners, with a high probability, $1 - \frac{1}{(\mathcal{M}_i)^{(k-1)m}}$, (depending on the range of r_i). Also, if any two numbers $\mathcal{M}_i < \mathcal{M}_j$, then$\mathcal{F}(\mathcal{M}_i) + r_i < \mathcal{F}(\mathcal{M}_j)$ will hold.

Step 4: Servers. Each server $\mathcal{S}_\phi$ executes the following operation:

$$input^{\mathcal{S}_\phi}[i] \leftarrow A(v)_i^{\phi}, 1 \leq i \leq m; output^{\mathcal{S}_\phi}[] \leftarrow \mathcal{PF}(input^{\mathcal{S}_\phi}[])$$

$\mathcal{S}_\phi$ collects additive shares from each DB owner and places them in an array (denoted by $input^{\mathcal{S}_\phi}[]$), on which $\mathcal{S}_\phi$ executes the permutation function $\mathcal{PF}$. Then, $\mathcal{S}_\phi$ sends the output the permutation function $output^{\mathcal{S}_\phi}[]$ to the announcer $\mathcal{S}_a$ that does the following:

$$foutput^{\mathcal{S}_a}[i] \leftarrow output^{\mathcal{S}_1}[i] + output^{\mathcal{S}_2}[i], 1 \leq i \leq m \tag{13}$$

$$max, index \leftarrow FindMax(foutput^{\mathcal{S}_a}[]) \tag{14}$$

$\mathcal{S}_a$ adds the i^{th} outputs received from $\mathcal{S}_1$ and $\mathcal{S}_2$, and compares all those numbers to find the maximum number (denoted by *max*). Also, $\mathcal{S}_a$ produces the index position (denoted by *index*) corresponding to the maximum number in $foutput^{\mathcal{S}_a}[]$. Finally, $\mathcal{S}_a$ creates additive secret-shares of *max* (denoted by $A(max^{\mathcal{S}_\phi})$, $\phi \in \{1, 2\}$), as well as, of *index* (denoted by $A(index)^{\mathcal{S}_\phi}$), and sends them to $\mathcal{S}_\phi$ that forwards such additive shares to DB owners. Note, if the *protocol does not require to reveal the identity* of the DB owner having the maximum value, $\mathcal{S}_a$ does not send additive shares of *index*.

Step 5a: DB owner. Now, the DB owners' task is to find the maximum value and/or the identity of the DB owner who has the maximum value. To do so, each DB owner performs the following:

$$\text{max} \leftarrow A(max)^{\mathcal{S}_1} + A(max)^{\mathcal{S}_2} \tag{15}$$

$$\text{index} \leftarrow A(index)^{\mathcal{S}_1} + A(index)^{\mathcal{S}_2}, pos \leftarrow \mathcal{RPF}(\text{index}) \tag{16}$$

The DB owner finds the identity of the DB owner having the max value by adding the additive shares and by implementing reverse permutation function $\mathcal{RPF}$. ($\mathcal{RPF}$ works since $\mathcal{PF}$ is known to DB owners and servers; see Assumptions given in §4). To find the max value, they implement $\mathcal{F}(z)$ and $\mathcal{F}(z+1)$ and evaluate $\mathcal{F}(z) \leq \text{max} < \mathcal{F}(z+1)$, where $z \in \{1, 2, \ldots\}$.[4] If this condition holds to be true, then z is the max value, and if $z = \mathcal{M}_i$, then $\mathcal{DB}_i$ knows that he/she holds the max value. Obviously, if $\mathcal{DB}_i$ does not hold the max value, then $\mathcal{M}_i < \mathcal{F}(\mathcal{M}_i) + r_i < \mathcal{F}(\mathcal{M}_i + 1) \leq \mathcal{F}(z) \leq \text{max}$.

Step 5b: DB owner. By the end of Step 5a, the DB owners know the max value and the identity of the DB owner having the same max value, due to *pos*. But, if there are more than one DB owners having the max value, the other DB owners cannot learn about it. The reason is: the server $\mathcal{S}_a$ can find only the max value, while, recall that, if more than one DB owners have the same max value, say $\mathcal{M}$, they produce a different value, due to using different random numbers in Step 3 (Equation 12). Thus, we need to execute this step 5b to know all DB owners having the max value. After comparing its max values against max, $\mathcal{DB}_i$ knows whether it possesses the maxi value or not. Depending on this, $\mathcal{DB}_i$ generates a value $\alpha_i = 0$ or $\alpha_i = 1$, creates additive shares of α_i, and sends to $\mathcal{S}_\phi$, $\phi \in \{1, 2\}$.

Step 6: Servers. Server $\mathcal{S}_\phi$ allocates the received additive shares to a vector, denoted by *fpos*, and sends the vector *fpos* to all DB owners, *i.e.*, $fpos^{\mathcal{S}_\phi}[i] \leftarrow A(\alpha)_i^{\mathcal{S}_\phi}, 1 \leq i \leq m$.

Step 7: DB owner. Each DB owner adds the received additive shares to obtain the vector fpos[].

$$\text{fpos}[i] \leftarrow fpos^{\mathcal{S}_1}[i] + fpos^{\mathcal{S}_2}[i], 1 \leq i \leq m \tag{17}$$

By fpos[], DB owners discover which DB owners have the maximum value, since, recall that in Step 5, $\mathcal{DB}_i$ that satisfies the condition ($\mathcal{F}(\mathcal{M}_i) \leq \text{max} < \mathcal{F}(\mathcal{M}_i + 1)$) requests $\mathcal{S}_\phi$ to place additive share of 1 at $fpos^{\mathcal{S}_\phi}[i]$.

Example 6.3.1. Refer to Tables 1-3, and consider that all hospitals wish to find the maximum age of a patient corresponding to the common disease and which hospitals treat such patients. Assume $\eta = 5003$ and that all hospitals know cancer as the common disease.

In Step 3, all hospitals, *i.e.*, DB owners, find their maximum values in the attribute Age corresponding to common disease and implement $\mathcal{F}(x) = x^4 + x^3 + x^2 + x + 1$, sent by the initiator.

$$\mathcal{F}(6) = 1555 + 216 = 1771 = (5000 - 3229) \bmod 5003$$
$$\mathcal{F}(8) = 4681 + 1 = 4682 = (5500 - 818) \bmod 5003$$
$$\mathcal{F}(8) = 4681 + 319 = 5000 = (2500 + 2500) \bmod 5003$$

Further, they add random numbers $(216, 1, 319)$ and create additive shares, which are outsourced to $\mathcal{S}_1$ and $\mathcal{S}_2$. In Step 4, $\mathcal{S}_1$ holds $\langle 5000, 5500, 2500 \rangle$, permutes them, and sends to $\mathcal{S}_a$. $\mathcal{S}_2$ holds $\langle -3229, -818, 2500 \rangle$, permutes them, and sends to $\mathcal{S}_a$.

$\mathcal{S}_a$ obtains $\langle 4682, 5000, 1771 \rangle$ by adding the received shares from $\mathcal{S}_1$, $\mathcal{S}_2$, and finds 5000 as the max value and 'Hospital 2' to which this value belongs. Finally, $\mathcal{S}_a$ creates additive shares of $5000 = (4000 + 1000) \bmod 5003$, additive shares of the identity of the DB owner: $2 = (200 - 198) \bmod 5003$, and sends to DB owners via $\mathcal{S}_\phi$.

In Step 5a, all hospitals will know the maximum value as 5000 (with random value added) and identity of the DB owner as 2 on which they implement the reverse permutation function to obtain the correct identity as 'Hospital 3'. Then, 'Hospital 1' knows that they do not hold the maximum, since $\mathcal{F}(6) + 216 < \mathcal{F}(7) < 5000$. 'Hospital 2' knows that they hold the maximum, since $\mathcal{F}(8) < 5000 < \mathcal{F}(9)$. Also, 'Hospital 3' knows that they hold the maximum. To know which hospitals have the maximum value, in Step 5b, Hospitals 1, 2, 3' create additive shares of 0, 1, 1, respectively, as: $0 = (200 - 200) \bmod 5003$, $1 = (300 - 299) \bmod 5003$, and $1 = (200 - 199) \bmod 5003$, and send to $\mathcal{S}_1$ and $\mathcal{S}_2$. Finally, in Step 6, $\mathcal{S}_1$ and $\mathcal{S}_2$ send $\langle 200, 300, 200 \rangle$ and $\langle -200, -299, -199 \rangle$ to all hospitals. In Step 7, hospitals add received shares, resulting in $\langle 0, 1, 1 \rangle$. It shows that 'Hospitals 2, 3' have the maximum value 8. ■

6.4 PSI Median Query

A PSI median query over cost column corresponding to disease column over Tables 1-3 returns $\{\langle \text{Cancer}, 300 \rangle\}$ (here, we first added the cost of treatment per disease at each DB owner). For PSI median, we extend the method of finding max by executing all steps as specified in §6.3 with an additional process in Step 2. Particularly, $\mathcal{S}_a$ in Step 2 of §6.3 after adding shares, sorts them, and finds the median value. If number of DB owners is odd (even), then $\mathcal{S}_a$ finds the middle (two middle) values in the sorted shares.

6.5 PSI Count Query

We extend PSI method (§5) to only reveal the count of common items among DB owners (*i.e.*, the cardinality of the common item), instead of revealing common items. Recall that servers $\mathcal{S}_\phi$ know a permutation function $\mathcal{PF}_{s1}$ that is not known to DB owners. The idea behind this is to find the common items over χ and to permute the final output at servers before sending the vector (of additive share form) to DB owners. Thus, when DB owners perform computation on the vector received from servers to know the final output, the position of one in the vector will not reveal common items, while the count of one will reveal the cardinality of the common items. Thus, PSI count method follows all steps of PSI as described in §5.1 with an addition of permutation function execution by servers before sending the output to DB owners.

[4] To reduce the computation cost, we can select number z using binary search method.

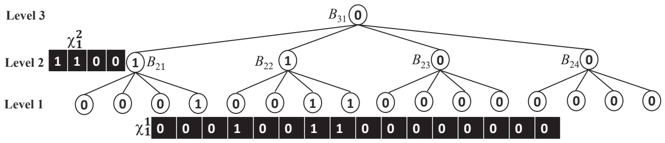


Figure 2: Bucket tree for 16 values.

6.6 Extending PSI over Multiple Attributes

In the previous sections, we explained PSI over a single attribute (or a set). We can trivially extend it to multiple attributes (or multisets). Particularly, such a query can be express in SQL as follows:

SELECT A_c, A_x FROM db_1 INTERSECT ... INTERSECT SELECT A_c, A_x FROM db_m

Recall that in PSI finding method §5.1, $\mathcal{DB}_j$ sends additive shares of a table χ_j of length $b = |\text{Dom}(A_c)|$, where A_c was the attributes on which we executed PSI. Now, we can extend this method by creating a table χ_j of length $b = |\Pi_{i>0}\text{Dom}(A_i)|$, where A_i are attributes on which we want to execute PSI. However, as the domain size and the number of attributes increase, such a method incurs the communication overhead. Thus, to apply the PSI method over a large (and real) domain size, as well as, to reduce the communication overhead, we provide a method, named as bucketization-based PSI.

Optimization: bucketization-based PSI. Before going to steps of this method, let us consider the following example:

Example 6.6.1. Consider two attributes A with $|\text{Dom}(A)| = 8$ and B with $|\text{Dom}(B)| = 2$. Thus, DB owners can create χ_j of 16 cells. Assume that there are two DB owners: $\mathcal{DB}_1$ with χ_1 whose only positions 4, 7, 8 have one; and $\mathcal{DB}_2$ with χ_2 whose only positions 1, 6, 8 have one. Thus, each DB owner sends/receives a vector of length 16 from each server. Now, to reduce communication, we create buckets over the cell of χ and build a tree, called *bucket-tree*, of depth $\log_\kappa |\chi|$, where κ is the number of the maximum number of child nodes that a node can have. Bucket-tree in created in a bottom-up manner, by non-overlap grouping of κ nodes. For each level of bucket-tree a hash table is created (similar to χ). Notation χ_j^i denotes this table for i^{th} level of bucket-tree at $\mathcal{DB}_j$, and $\chi_j^i[k] = 1$, if k^{th} node at the i^{th} level has 1.

Figure 2 shows bucket-tree for $\mathcal{DB}_j$, $|\chi| = 16$, and $\kappa = 4$, with appropriate one and zero in χ_1^i. Note that the second level shows four nodes $B_{21}, B_{22}, B_{23}, B_{24}$ corresponding to $1-4$, $5-8$, $9-12$, and $13-16$. Since $\mathcal{DB}_1$ has one at 4, 7, 8 leaf nodes, we obtain $\chi_1^2 = \langle 1, 1, 0, 0 \rangle$, *i.e.*, $B_{21} = 1, B_{22} = 1, B_{23} = 0, B_{24} = 0$. Here, $B_{21} = 1$, since one of its child nodes has one. Now, when computing PSI, $\mathcal{DB}_j$ starts the computation shown in Step 2 of §5.1 over the specified i^{th} levels' χ_j^i. The computation is continued only for the child nodes, whose parent nodes resulted in one in Step 3 of §5.1.

For example, in Figure 2, $\mathcal{DB}_j$ can execute PSI for χ_j^2 and know that the only desired bucket nodes are B_{21} and B_{22} that contain common items. Thus, in the next round, they execute PSI over the first eight items of χ_j^1, *i.e.*, child nodes of B_{21} and B_{22}. Hence, while we use two communication rounds, DB owners/servers send 4+8=12 numbers instead of 16 numbers. ■

Bucketization-based PSI has the following steps:

Step 1a: DB owner. Build the tree as specified in Example 6.6.1.

Step 1b: DB owner. Outsource additive shares of i^{th} level's χ_j^i.

Step 2: Servers. Servers compute PSI using Step 2 of §5.1 over χ_j^i $(1 \leq j \leq m)$ and provide answers to DB owners.

Step 3: DB owner. $\mathcal{DB}_j$ computes results to find the common items in χ_j^i and discards all non-common values of χ_j^i and their child nodes. $\mathcal{DB}_j$ requests servers to execute the above Step 2 for χ_j^{i-1} that has values corresponding to all non-discarded nodes of $(i-1)^{th}$ level node. **Note:** The role of DB owners in traversing the tree (*i.e.*, the above Step 3) can be eliminated by involving $\mathcal{S}_a$.

Open problem. In bucketization, we perform PSI at layers of the tree to eliminate ranges where corresponding child nodes have zero. However, if the data is dense (*i.e.*, data covers most of the domain values), then bucketization-based PSI may incur overhead, since all nodes in the tree may correspond to one, leading to PSI execution on all those nodes including leaf nodes. In contrast, if the data is sparse (*i.e.*, the domain is much larger than the data, as is the case of the domain to be a cartesian product of domains of two or more attributes), then higher-level nodes in the tree may have 0, leading to eliminate ranges of the domain on which PSI is performed. Developing an optimal bucketization strategy that minimizes PSI execution is an interesting open problem.

7 PRIVATE SET UNION (PSU) QUERY

This section develops a method for finding the union of values among $m > 1$ different DB owners over an attribute A_c.

High-level idea. Likewise PSI method (as given in §5), each DB owner uses a publicly known hash function to map distinct values of A_c attribute in a table of cells at most $|\text{Dom}(A_c)|$, where $|\text{Dom}(A_c)|$ refers to the size of the domain of A_c, and outsources each element of the table in additive share form to *two servers* $\mathcal{S}_\phi$, $\phi \in \{1, 2\}$. $\mathcal{S}_\phi$ computes the union obliviously, thereby DB owners will receive a vector of length $|\text{Dom}(A_c)|$ having either 0 or 1 of additive shared form. After adding the share for an i^{th} element, DB owners only know whether the element is in the union or not; nothing else.

Step 1: DB owner. This step is identical to Step 1 of PSI (§5.1).

Step 2: Server. Server $\mathcal{S}_\phi$ holds the ϕ^{th} additive share of the table χ of m DB owners and executes the following operation:

$$rand[] \leftarrow \mathcal{PRG}(seed)$$
$$output_i^{\mathcal{S}_\phi} \leftarrow ((\textstyle\sum_{j=1}^{j=m} A(x_i)_j^\phi) \times rand[i]) \bmod \delta \tag{18}$$

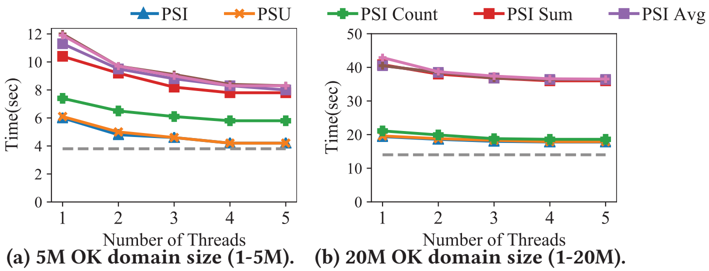
Server $\mathcal{S}_\phi$: (*i*) generates b pseudorandom numbers that are between 1 and $\delta - 1$, (*ii*) performs (arithmetic) addition of the i^{th} additive secret-shares from all DB owners, (*iii*) multiplies the resultant of the previous step with i^{th} pseudorandom number and then takes modulo, and (*iv*) sends b results to all DB owners.

Step 3: DB owner. On receiving two vectors, each of length b, from two servers, DB owners execute modular addition over i^{th} shares of both vectors to get the final answer (Equation 19). It results in either zero or a random number, where zero shows that i^{th} element of χ is not present at any DB owner, while a random number shows i^{th} element of χ is present at one of the DB owners.

$$fop_i \leftarrow (output_i^{\mathcal{S}_1} + output_i^{\mathcal{S}_2}) \bmod \delta \tag{19}$$

8 EXPERIMENTAL EVALUATION

This section evaluates the scalability of Prism on different-sized datasets and a different number of DB owners. Also, we compare Prism against other MPC-based systems. We used a 16GB RAM machine with 4 cores for each of the DB owners and three AWS



(a) 5M OK domain size (1-5M). (b) 20M OK domain size (1-20M).

Figure 3: Exp 1. Prism performance on multi-threaded implementation at AWS.

Real data column					For verification					Average
OK	PK	LN	SK	DT	vOK	vPK	vLN	vSK	vDT	aOK

Table 11: Table structure created by Prism.

servers of 32GB RAM, 3.5GHz Intel Xeon CPU with 16 cores to store shares. Communication between DB owners and servers were done using the scp protocol, and η, δ were 227, 113, respectively.

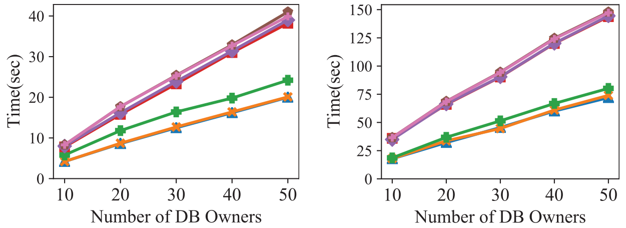
8.1 Prism Evaluation

Dataset generation. We used five columns (Orderkey (OK), Partkey (PK), Linenumber (LN), Suppkey(SK), and Discount (DT)) of LineItem table of TPC-H benchmark. We experimented with domain sizes (*i.e.*, the number of values) of 5M and 20M for the ***OK column*** on which we executed PSI and PSU. Further, we selected at most ***50 DB owners***. To our knowledge, this is the first such experiment of multi-owner large datasets. OK column is used for PSI/PSU, and other columns were used for aggregation operations. To generate secret-shared dataset, each DB owner maintained a LineItem table containing at most 5M (20M) OK values. To outsource the database, each DB owner did the following:

(1) Created a table of 11 columns, as shown in Table 11, in which the first five columns contain the secret-shared data of LineItem table, the next five columns contain the corresponding verification data, and the last column (aOK) was used for computing the average. All verification column names are prefixed with the character 'v.'
(2) First column of Table 11 was created over OK column of LineItem table (using Step 1 of §5.1) for executing PSI/PSU over OK. vOK column was created to verify PSI results (using Step 1 of §5.2).
(3) Columns PK and vPK were created using the following command: `select OK, sum(PK) from LineItem group by OK`. Other columns ⟨LN, SK, DT, vLN, vSK, vDT⟩ were created by using similar SQL commands. Column aOK was created using the following command: `select count(*) from LineItem group by OK`.
(4) Finally, permuted all values of all verification columns and create additive shares of ⟨OK and vOK⟩, as well as, multiplicative shares of all remaining columns.

Share generation time. The time to generate two additive shares and three multiplicative shares of the respective first five columns of Table 11 in the case of 5M (or 20M) OK domain size was 121s (or 548s). The time for creating each additional column for verification took 20s (or 90s) in the case of 5M (or 20M) domain values.

Exp 1. Prism performance on multi-threaded implementation at AWS. Since identical computations are executed on each row of the table, we exploit multiple CPU cores by writing Prism's the parallel implementation that divides rows into multiple blocks with each thread processing a single block. We increased the number of threads from 1 to 5; see Figure 3, while fixing DB owners to 10. Increasing threads more than 5 did not provide speed-up, since reading/writing of data quickly becomes the bottleneck as the number of threads increase. Observe that the data fetch time from the database remains (almost) identical; see Figure 3.



(a) 5M OK domain size (1-5M). (b) 20M OK domain size (1-20M).

Figure 4: Exp 2. Prism dealing with multiple DB owners.

PSI and PSU queries. Figure 3 shows the time taken by PSI/PSU over OK column. Observe that as the number of values in OK column increases (from 5M to 20M), the time increases (almost) linearly from 4s to 18s, respectively.

Aggregation queries over PSI. We executed PSI count, average, sum, maximum, and median queries; see Figure 3. Observe that the processing time of PSI count is almost the same as that of PSI, since it involves only one round of computation in which we permute the output of PSI. In contrast, other aggregation operations (sum, average, maximum, and median) incur almost twice processing cost at servers, since they involve computing PSI over OK column in the first round and, then, computing aggregation in the second round. For this experiment, we computed the sum only over DT column and maximum/median over PK column. Table 12 shows the impact of computing sum and maximum over multiple attributes (from 1 to 4). As we increase the number of attributes, the computation of respective aggregation operations also increases, due to additional addition/multiplication/modulo operations on additional attributes.

Data size	**Sum over different attributes**				**Max over different attributes**			
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
5M	8.2	12.1	15.9	20.4	10	14.6	19	23.5
20M	33.4	48.6	63.5	81.9	36.6	53.3	70	87.4

Table 12: Exp 1. Multi-column aggregation query performance (time in seconds).

Exp 2. Impact of the number of DB owners. Prism deals with multiple DB owners; thus, we investigated the impact of DB owners by selecting 10, 20, 30, 40, 50 DB owners, for two different domain sizes of OK column. Figure 4 shows the server processing time for PSI, PSU, and aggregation over PSI. Observe that as the number of DB owners increases, the computation time at the server increases linearly, due to the linearly increasing number of addition/multiplication/modulo operations; *e.g.*, on 5M OK values, PSI processing took 4.2s, 8.6s, 12.5s, 16.2s, and 20s in the case of 10, 20, 30, 40, 50 DB owners.

Exp 3. DB owner processing time in result construction. In Prism, DB owners perform computation on additive or multiplicative shares. Table 14 shows the processing time at a DB owner over 5M and 20M domain values for different operations. It is clear that the DB owner processing time is significantly less than the server processing time. In case of 5M (20M) OK values and 50 DB owners,

Papers	[39] & [45]	[51]	[3]	[2]	[37]	[38]	Jana [5]†	SMCQL [6]	Sharemind [8]	Conclave [54]‡	**PRISM**
Operations supported	PSI	PSI	PSI	PSI	PSI	PSI	PSI, PSU, aggregation	PSI via join & aggregation	PSI via join & aggregation	PSI via join & aggregation	**PSI, PSU, aggregation**
Verification Support	×	×	×	✓	✓	×	×	×	×	×	✓
Scalability based on experiments reported (dataset size & time)	N/A	32768 (≈50 **m**)	1 million (≈2 **h**)	32768 (≈16 **m**)	1 billion (≈10 **m**)	1000 (≈9 **m**)	1 million (≈1 **h**)	>23 million (≈23 **h**)	30000 (>2 **h**)	4 million (8 **m**)	**20 million (At most 8 s)**
Communication among servers	N/A	N/A	N/A	N/A	N/A	N/A	**Yes** ∗	**Yes** ∗	**Yes** ∗	**Yes** ∗	**No**
Computational Complexity	$O(n^m)$	$O(\alpha mn)$	$O(n^m)$	$O(mn^2)$	$O(mn)$ ‡‡	$O(n^m)$	$O(n^m)$	N/A ∗	$O(n^m)$	N/A ∗	$O(mX)$

Table 13: Comparison of existing *cloud-based* techniques against PRISM. Notes. (*i*) The **scalability numbers are taken from the respective papers.** (*ii*) Results of Sharemind [8] are taken from Conclave [54] experimental comparison. (*ii*) #DB owners were in each paper was reported two; thus, we executed PRISM for two DB owners for this table. (*iv*) Only Jana, SMCQL, Sharemind, and Conclave provide identical security like PRISM. (*v*) **h**: hours. **m**: minutes. **s**: seconds. †: We setup Jana for two DB owners each with 1M values in our experiments. ‡: Conclave [54] uses a trusted party. **Yes**: Requires communication among servers. **No**: No communication among servers. ∗: Based on MPC-based systems. ∗∗: N/A because executing operation in cleartext or at the trusted party. m: #DB owners. n: DB size. X: domain size. ‡‡: A insecure technique that reveals the size of the intersection, and hence fast. α: The cost of Bilinear Map pairing technique.

Data Size	**5M**	**20M**
PSI	1.3	4.8
Count	1.7	5.4
Sum	3.1	10.3
Avg	3.2	10.3
Max	2.8	9.5
PSU	1.3	4.8

Table 14: Exp 3. DB owner processing time in result construction (in seconds).

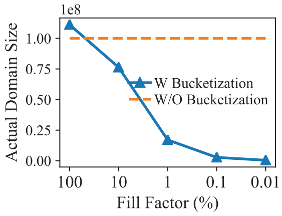


Figure 5: Exp 4. Impact of bucketization.

each DB owner took at most 4s (13s) in PSI Sum (PSI Sum) query, while servers took at least 20s (72s) in PSI (PSI) query; see Figure 4.

Exp 4. Impact of bucketization. Figure 5 shows the reduction in the number of values on which we need to execute PSI when using bucketization technique (§6.6). We created a tree with fanout 10, height 9, and 100M values at the leaf level. In Figure 5, we refer to the percentage of leaf nodes of the tree that containing one as *fill factor*. We use a term *actual domain size* (in Figure 5) that refers to the number of items on which we execute PSI. The actual domain size is different from the *real domain size* that refers to domain values given to us, *i.e.*, 100M. The actual domain size depends on the fill factor and impacts the performance of PSI. When fill factor is 100% (*i.e.*, all leaf nodes have one; thus, the entire tree has one), the actual domain size was 111M. But, if the fill factor was only 0.01% of 100M values (*i.e.*, 10K), then most of the tree contained zero; thus, we run PSI only on actual domain size of 400K, instead of real domain size of 100M. Note, for this experiment, we generated the data randomly. If there is a correlation in the data (the case in most real-world datasets), bucketization results will be even better.

8.2 Comparing with Other Works

We compare PRISM against the state-of-the-art cloud-based industrial MPC-based systems: Galois Inc.'s Jana [5], since it provides identical security guarantees at servers as PRISM. To evaluate Jana, we inserted two LineItem tables (each of 1M rows) having ⟨OK, PK, LN, SK, DT⟩ columns and executed join on OK column. However, the join execution took more than 1 hour to complete.

[2, 3, 37–39, 45, 51] provide **cloud-based PSI/PSU/aggregation** techniques/systems. ***We could not experimentally compare PRISM against such systems***, since none are open-source, except SMCQL [6], (which we installed and works for a very small data and incurs runtime errors). Thus, in Table 13, we report experimental results from those papers, just for intuition purposes. With the exception of [37], none of the techniques supports a large-sized dataset, has quadratic/exponential complexity, or uses expensive cryptographic techniques [51]. While [37] scales better, it does not support aggregation and, also, reveals which item is in the intersection set. For a fair comparison, we report PRISM results only for two DB owners in Table 13, since other papers do not provide experimental results for more than two DB owners. In our experiments (Figure 4a), PRISM supports 50 DB owners and takes at most ≈41 seconds on 5M values. Also, note that, in case of 1B values and two DB owners, PRISM takes ≈ 7.3mins, unlike [37] that took ≈10mins.

Several **non-cloud-based PSI approaches** also exist and ***cannot be directly compared against*** PRISM, due to a different model used (in which DB owners communicate amongst themselves and do not outsource data to cloud) and/or different security properties (*e.g.*,[4, 15, 21, 23, 26, 32, 41, 43, 47, 49]). A *survey of PSI* protocols may be found in [49]. Many schemes including Yao's approach [57] for comparison/max finding were proposed; *e.g.*, [9–11, 20, 30, 48, 53]. Such techniques have limitations: many communication rounds, restricted to two DB owners, quadratic computation cost at servers, not dealing with malicious adversaries in cloud settings, and/or no support for result verification.

Comparison between PRISM and OBSCURE [28]. While both PRISM and OBSCURE are based on secret-sharing, they are significantly different from each other in terms of: (*i*) purposes: PRISM is for computing simple aggregation over PSI/PSU queries over multi-owner databases, while OBSCURE is for complex conjunctive/disjunctive aggregation query processing over outsourced data and does not support PSI/PSU queries; (*ii*) implementation: PRISM is based on domain-based representation, while OBSCURE is based on unary representation; (*iii*) query execution complexities: PRISM complexity is $O(m \times \mathrm{Dom}(A_c))$, where m is #DB owners and $\mathrm{Dom}(A_c)$ is the domain of attribute A_c, while OBSCURE complexity is $O(n \times L)$, where n is the number of tuples and L is the length of a value in unary representation. Thus, ***a direct comparison between the two non-identical systems is infeasible***. Full version [1] shows overheads of these different secret-sharing techniques.

9 CONCLUSION

This paper describes PRISM based on secret-sharing that allows multiple DB owners to outsource data to (a majority of) non-colluding servers, behaving like honest-but-curious and malicious servers in terms of computations that they perform. PRISM exploits the additive and multiplicative homomorphic property of secret-sharing techniques to implement set operations (intersection, union) and aggregation functions. Experimental results show PRISM scales to both a large number of DB owners and to large datasets.

REFERENCES

[1] Full version of this paper is available at: https://isg.ics.uci.edu/publications/.

[2] A. Abadi et al. VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In *FC*, pages 149–168, 2016.

[3] A. Abadi et al. Efficient delegated private set intersection on outsourced private datasets. *IEEE Trans. Dependable Secur. Comput.*, 16(4):608–624, 2019.

[4] T. Araki et al. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*, pages 805–817, 2016.

[5] D. W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.

[6] J. Bater et al. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.

[7] M. Blum et al. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984.

[8] D. Bogdanov et al. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[9] D. Bogdanov et al. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *NordSec*, pages 59–74, 2014.

[10] M. Burkhart et al. Fast privacy-preserving top-k queries using secret sharing. In *ICCCN*, pages 1–7, 2010.

[11] M. Burkhart et al. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–240, 2010.

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

[13] R. Canetti et al. Adaptively secure multi-party computation. In G. L. Miller, editor, *STOC*, pages 639–648, 1996.

[14] D. Cash et al. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[15] H. Chen et al. Fast private set intersection from homomorphic encryption. In *CCS*, pages 1243–1255, 2017.

[16] J. H. Cheon et al. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 95-A(8):1366–1378, 2012.

[17] K. Chida et al. An efficient secure three-party sorting protocol with an honest majority. *IACR Cryptol. ePrint Arch.*, 2019:695, 2019.

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

[19] E. D. Cristofaro et al. Fast and private computation of cardinality of set intersection and union. In *CANS*, pages 218–231, 2012.

[20] I. Damgård et al. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[21] C. Dong et al. When private set intersection meets big data: an efficient and scalable protocol. In *CCS*, pages 789–800, 2013.

[22] R. Egert et al. Privately computing set-union and set-intersection cardinality via bloom filters. In *ACISP*, pages 413–430, 2015.

[23] M. J. Freedman et al. Efficient private matching and set intersection. In *EUROCRYPT*, pages 1–19, 2004.

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

[25] O. Goldreich et al. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[26] O. Goldreich et al. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[27] D. M. Goldschlag et al. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[28] P. Gupta et al. Obscure: Information-theoretic oblivious and verifiable aggregation queries. *Proc. VLDB Endow.*, 12(9):1030–1043, 2019.

[29] H. Hacigümüs et al. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[30] K. Hamada et al. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, pages 202–216, 2012.

[31] C. Hazay et al. Scalable multi-party private set-intersection. In *PKC*, pages 175–203, 2017.

[32] Y. Huang et al. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

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

[34] M. Ion et al. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. *IACR Cryptol. ePrint Arch.*, 2019:723, 2019.

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

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

[37] S. Kamara et al. Scaling private set intersection to billion-element sets. In *FC*, pages 195–215, 2014.

[38] F. Kerschbaum. Collusion-resistant outsourcing of private set intersection. In *SAC*, pages 1451–1456, 2012.

[39] F. Kerschbaum. Outsourced private set intersection using homomorphic encryption. In *ASIACCS*, pages 85–86, 2012.

[40] L. Kissner et al. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.

[41] V. Kolesnikov et al. Practical multi-party private set intersection from symmetric-key techniques. In *CCS*, pages 1257–1272, 2017.

[42] P. H. Le et al. Two-party private set intersection with an untrusted third party. In *CCS*, pages 2403–2420, 2019.

[43] Y. Li et al. Delegatable order-revealing encryption. In *AsiaCCS*, pages 134–147, 2019.

[44] Y. Lindell. Secure multiparty computation (MPC). *IACR Cryptol. ePrint Arch.*, 2020:300, 2020.

[45] F. Liu et al. Encrypted set intersection protocol for outsourced datasets. In *ICCE*, pages 135–140, 2014.

[46] S. Madden et al. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[47] D. Many et al. Fast private set operations with sepia. *ETZ G93*, 2012.

[48] T. Nishide et al. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, pages 343–360, 2007.

[49] B. Pinkas et al. Faster private set intersection based on OT extension. In *USENIX Security*, pages 797–812, 2014.

[50] B. Pinkas et al. SpOT-Light: Lightweight private set intersection from sparse OT extension. In *CRYPTO*, pages 401–431, 2019.

[51] S. Qiu et al. Identity-based private matching over outsourced encrypted datasets. *IEEE Trans. Cloud Comput.*, 6(3):747–759, 2018.

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

[53] J. Vaidya et al. Privacy-preserving top-k queries. In *ICDE*, pages 545–546, 2005.

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

[55] C. Wang et al. Secure ranked keyword search over encrypted cloud data. In *ICDCS*, pages 253–262, 2010.

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

[57] A. C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

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

[59] Q. Zheng et al. Verifiable delegated set intersection operations on outsourced encrypted data. In *IC2E*, pages 175–184, 2015.