
# SecretFlow-SCQL: A Secure Collaborative Query pLatform

Wenjing Fang*
Ant Group

Shunde Cao
Ant Group

Guojin Hua
Ant Group

Junming Ma
Ant Group

Yongqiang Yu
Ant Group

Qunshan Huang
Ant Group

Jun Feng
Ant Group

Jin Tan
Ant Group

Xiaopeng Zan
Ant Group

Pu Duan
Ant Group

Yang Yang
Ant Group

Li Wang
Ant Group

Ke Zhang
Ant Group

Lei Wang
Ant Group

## ABSTRACT

In the business scenarios at Ant Group, there is a rising demand for collaborative data analysis among multiple institutions, which can promote health insurance, financial services, risk control, and others. However, the increasing concern about privacy issues has led to data silos. Secure Multi-Party Computation(MPC) provides an effective solution for collaborative data analysis, which can utilize data value while ensuring data security. Nevertheless, the performance bottlenecks of MPC and the strong demand for scalability pose great challenges to secure collaborative data analysis frameworks.

In this paper, we build a secure collaborative data analysis system SCQL with a general purpose. We design more efficient MPC protocols and relational operators to meet the demand for scalability. In terms of system design, we aim to implement a system with security, usability, and efficiency.

We conduct extensive experiments on SCQL to validate our optimization improvements: (1) Our optimized secure sort protocol sorts one million 64-bit data in only 4.5 minutes, 126× faster than EMP(9.4 hours). (2) The end-to-end execution time of the typical vertical scenario query is reduced by 1991× from the state-of-the-art semi-honest collaborative analysis framework Secrecy(rewritten with Additive Secret Sharing protocol), with appropriate security trade-offs. (3) We test the system in the WAN setting with $input\ size = 10^7$ to demonstrate the scalability. We have successfully deployed SCQL to address problems in real-world business scenarios at Ant Group.

**PVLDB Reference Format:**
Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, Yang Yang, Li Wang, Ke Zhang, and Lei Wang. SecretFlow-SCQL: A Secure Collaborative Query pLatform. PVLDB, 17(12): 3987 - 4000, 2024.
doi:10.14778/3685800.3685821

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/secretflow/scql.

*fangwenjing_scu@126.com

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685821

## 1 INTRODUCTION

In recent years, there has been a rising demand for *collaborative data analysis*, especially in the fields of medical research, financial services, and risk management [3, 28, 31, 43, 50]. It is proved that the supplementary features or samples provided by different institutions can bring further improvement in performance. However, with the increasing concern for privacy issues and the introduction of relevant regulations (e.g., GDPR [1]), the data between different institutions cannot be shared, resulting in data silos. As a typical scenario, when a user submits a medical insurance claim, the insurance company needs to combine external medical data for claim review. However, the hospitals cannot disclose their databases, which violates patient confidentiality. Essentially, we need an infrastructure to enable arbitrary queries to be securely executed on the collective data of mutually distrustful institutions.

The most straightforward solution to the aforementioned problem is to utilize a Trusted Execution Environment (TEE) to protect the security of the data, such as works like Opaque [57], EnclaveDB [48], ObliDB [17], StealthDB [54] and OCQ [14]. TEE-based methods first send encrypted sensitive data to a hardware-based trust zone. Then the data is decrypted and processed within this trusted enclave. Because the data is processed in plaintext, there will be no performance loss in computations and no need for tedious algorithm adaptation. However, these methods require the deployment of specialized hardware and demand that users fully trust the hardware provider, which may not be satisfied in many practical applications. In addition, TEE-based methods are also vulnerable to various side-channel attacks [10, 33, 53].

Another longstanding research direction is encrypted databases, such as CryptDB [47], Monomi [52], BlindSeer [44], Arx [45] and Seabed [42]. Users encrypt the data and outsource it to servers, which directly perform computations on the encrypted data. These cryptography-based systems have to employ various cryptographic primitives to deal with different types of queries. For example, partially homomorphic encryption (PHE) [41] supports queries with aggregation, order preserving encryption (OPE) [8, 9] supports range queries, deterministic encryption (DET) [7] and searchable encryption (SE) [13, 51] supports match queries. Nevertheless, these property-preserving encryption schemes leak certain data properties ( e.g., the order and equality properties) and search patterns, which suffer from various attacks [11, 30, 32, 40]. Recent work [49] has attempted to solve the aforementioned problem by utilizing Turing-complete Fully Homomorphic Encryption (FHE) [21] to support various types of queries. However, there is still a long way to

go in terms of implementing general-purpose query operators (e.g., GROUP-BY ) and achieving a practical performance.

Secure Multi-Party Computation (MPC) [18], another cryptographic technique, offers a richer set of underlying primitives and better performance compared to FHE, which makes it another emerging solution for collaborative data analysis. MPC enables a group of independent data owners who do not trust each other to jointly compute a function that depends on all of their private inputs. MPC protocols ensure that each party learns nothing except the output after they participate in the computation. Recently, a series of works have attempted to design secure relational operators to achieve general-purpose collaborative queries and make efforts from different perspectives to improve the security and performance of queries.

SMCQL [3] is the first attempt to utilize MPC techniques, specifically Garbled Circuits [22], in the field of collaborative data analysis applied to healthcare scenarios. It decomposes the query plan and allows the data holders to perform local computations on plaintext data, which reduces time-consuming MPC calls and the computational scale involved. Conclave [55] improves efficiency further based on SMCQL. On the one hand, it utilizes Spark [56] to accelerate local plaintext computation. On the other hand, it designs hybrid MPC-plaintext protocols, which require selectively trusted party (STP) annotations, as alternatives to the MPC protocols. Senate [46] focuses on the scenario with malicious adversaries, where participants may arbitrarily deviate from the prescribed protocol in an attempt to violate data security. It decomposes the joint protocol of all participants into step-wise merging of smaller sub-circuits between two participants, thereby enhancing efficiency through parallelism. However, providing malicious security incurs significant additional costs. Secrecy [34] addresses another scenario where some data owners lack computational capabilities by outsourcing computation to three computing nodes. It designs secret relational operators and logical transformations to improve efficiency. Similarly, Scape [24] adopts the same outsourcing setup as Secrecy and it further provides guarantees for malicious security. It proposes optimized general join and aggregation operators, which are the core bottlenecks of query performance.

Although these works have taken a step forward in the direction of MPC-based collaborative data analysis, there are still some concerns when it comes to practical applications:

- SMCQL, Conclave, and Senate employ GC as the MPC backend. Although these techniques require only a small constant number of communication rounds, the actual communication and computation overhead is high, which limits their practical application compared to Secret Sharing (SS) schemes in terms of overall performance [16].
- Secrecy and Scape adopt the outsourced computation, which has two issues. Firstly, data owners often do not accept data being distributed to external computing nodes due to concerns about business competition and data privacy. Secondly, the end-to-end MPC mode cannot leverage data distribution for performance optimization, just like the optimizations in SMCQL and Conclave.
- Conclave, Secrecy, and Scape do not support SQL syntax, which is not user-friendly for data analysts.
- Real-world applications require support for larger data volumes and higher efficiency.

In this paper, we establish Secure Collaborative Query pLatform (SCQL), which enables mutually distrusting parties to make queries of their collective data and has comprehensive support for SQL operators. The Additive Secret Sharing scheme (cf. section 2) is employed to ensure the security of the computation process. The data owners serve as the computation nodes of the MPC, thereby keeping the data in the hands of its source.

To run the queries, we need to support the relational operators in SQL based on Additive Secret Sharing. Most importantly, we redesign and optimize the sorting protocol, which implements the GROUP-BY operator and is the bottleneck of query execution. Most existing implementations use sorting networks, which rely on expensive comparison operators. Instead, we employ the radix sort, which is more MPC-friendly.

Existing works demonstrate that data distribution can be used to improve the efficiency of collaborative data analysis. Typically, there are two types of data distribution: horizontal scenario and vertical scenario (cf. section 2). Existing optimizations allow for scalable queries in horizontal scenarios, but they have limited effectiveness in vertical scenarios. To fill this gap, we design more efficient relational operators specifically tailored for the security requirements in vertical scenarios. By making suitable security compromises, we greatly improve the query performance.

In terms of usability, SCQL establishes a virtual database and users can write SQL queries as if they were using a centralized database. Users do not need to worry about the physical distribution of data or be aware of the MPC expertise. SCQL automatically translates a query into the corresponding MPC protocols and delegates the computation to the underlying MPC infrastructure.

Besides, we propose the Column Control List (CCL) annotation (cf. section 4.3) in SCQL to describe users' security requirements. On the one hand, CCL is used for static semantic checks and rejects some illegal queries. On the other hand, it also defines the intermediate information that users allow to open, thus the system can automatically select a more efficient implementation at runtime.

Our contributions are summarized as follows:

- We design new MPC building blocks, e.g. shuffle and sort, which greatly alleviates the bottleneck of collaborative data analysis. Additionally, they may be of broader interest beyond SCQL and work in other MPC applications.
- We innovatively propose high-performance relational operators in vertical scenarios. To the best of our knowledge, this is the first work that delves into algorithm optimization specifically designed for secure collaborative data analysis in vertical scenarios.
- We implement and open-source a user-friendly and efficient framework, which is thoroughly evaluated and confirmed to provide significant improvements in performance at the levels of MPC building block (section 5.2), relational operators (section 5.3), and end-to-end queries (section 5.4).

The rest of this paper is organized as follows: we first introduce the basic settings and Additive Secret Sharing background in section 2. Then we explain the algorithm design in section 3, including the redesign of the sorting protocol to implement the oblivious

baseline and a detailed explanation of the optimization of relational operators in the vertical scenarios. In section 4, we provide a detailed introduction to the implementation. To provide a better explanation, we used an example to demonstrate the overall workflow and the use of CCL annotations. Further, in section 5, we conduct extensive experiments on SCQL to validate our optimization improvements. Finally, we briefly introduce related works in section 6 and conclude in section 7 .

## 2 PRELIMINARY

**Data Setting.** In collaborative data analysis, there are two types of data distribution: one is the *horizontal scenario*, where the participants share the same data schema but have different users, and the joint dataset is like putting them together horizontally. The other is the *vertical scenario*, where the participants share some common users, but each participant has different data columns, and the joint dataset is like putting them together vertically. We use two query examples to demonstrate these scenarios.

In medical analysis, horizontal cases are more common. The *Comorbidity* query in SMCQL [3] is shown in Figure 1, and it finds the top ten most common diagnoses for individuals with clostridium difficile (cdiff). In this case, hospitals hold data tables with the same schema, but their patients are different. The tables across the hospitals are merged with `UNION`. The algorithm optimization for horizontal scenarios is more straightforward. For example, in this case, each hospital can locally aggregate a partial statistic, and then only a small amount of joint computation is needed to merge these partial statistics. The optimizations in prior works [3, 55] are sufficient for horizontal scenarios.

```
SELECT
  diag
  , COUNT(*) as cnt
FROM (
    SELECT diag FROM P1.cdiff_cohort_diagnoses
    UNION ALL
    SELECT diag FROM P2.cdiff_cohort_diagnoses
)AS m
GROUP BY diag
ORDER BY cnt DESC
LIMIT 10
```

**Figure 1: Example query in horizontal scenario**

Figure 2 shows an example of a vertical scenario. Alice, a financial institution, holds the income, age, and credit rank of users; while Bob, an e-commerce platform, obtains transaction amount, and activity status of users. Alice wants to analyze the young people (ages between 20 and 30) and calculate the user count, average income, and maximum transaction amount with different credit ranks and active states on the e-commerce platform. We get the SQL statement in Figure 2 to meet the above requirements. In this case, participants hold different features, and user alignment is achieved through an `INNER JOIN` based on field *ID*. We will focus on performance optimization in this scenario later.

**Security Setting.** In joint analysis, participants need to reach a consensus on the queries to be executed, thus making the queries public. So protecting the queries is not within our security objectives.

In practice, the original data is prohibited from leaving its owner, e.g. the sensitive financial data and user privacy data. Therefore, we

```
SELECT
    ta.credit_rank
    , tb.is_active
    , COUNT(*) as user_cnt
    , AVG(ta.income) as avg_income
    , MAX(tb.order_amount) as max_amount
FROM ta INNER JOIN tb
ON ta.ID = tb.ID
WHERE ta.age >= 20
    AND ta.age <= 30
GROUP BY ta.credit_rank, tb.is_active
```

```
Alice: ta
+-------------+--------+
|    Field    |  Type  |
+-------------+--------+
| id          | string |
| credit_rank | int    |
| income      | int    |
| age         | int    |
+-------------+--------+
Bob: tb
+-------------+--------+
|    Field    |  Type  |
+-------------+--------+
| id          | string |
| order_amount| float  |
| is_active   | int    |
+-------------+--------+
```

**Figure 2: Example query in vertical scenario**

abandon the commonly adopted outsourcing computation mode which requires distributing data to computing nodes, even if the computation security is ensured by the trusted hardware or the MPC cluster. Instead, the participants should be both data holders and computing nodes.

We adopt Secret Sharing techniques to realize multi-party secure computation. Secret sharing techniques are more flexible than Garbled Circuits in implementing complex operators and achieve better performances in general. Efficient ABY3 protocol [37] is a popular choice among current collaborative analysis solutions [24, 34]. ABY3 is a three-party computation protocol, where the parties do not collude. In fact, any two parties can recover the third party's private data, and this violates the requirement of data not leaving its source. Hence, we employ Additive Secret Sharing in the semi-honest adversarial setting without non-collusion assumption, where the parties attempt to infer sensitive information while following the protocol.

Following the *MPC with pre-processing* paradigm [29, 38], we introduce a trusted third party (TTP) to generate correlated randomness, which enables more efficient and scalable MPC [6]. Note that it is only responsible for randomness generation and has no access to sensitive data of joint analysis. In theory, input-independent randomness can be massively pre-produced in the offline stage, thereby achieving a better online performance.

Although we present the Additive Secret Sharing for subsequent algorithm design, SCQL supports switching between different MPC infrastructures and schemes. It reserves the freedom for users to choose protocols based on their own security needs.

**Additive Secret Sharing.** Additive Secret Sharing shares $x$ additively in the ring $\mathbb{Z}_{2^l}$ as the sum of $n$ values, where $l$ is the bitwidth and $n$ is the number of parties. Though it naturally supports multiple parties, we introduce basic protocols under the 2PC setting.

- *Sharing Scheme*: $x = \langle x \rangle_0 + \langle x \rangle_1 \ mod \ 2^l$, where $\langle x \rangle_0$ and $\langle x \rangle_1$ are two random shares for secret $x$ and owned by $P_0$ and $P_1$ respectively. Note that we omit $2^l$ for conciseness in the following and denote a shared value in angle brackets.
- *Addition*: $\langle z \rangle = \langle x \rangle + \langle y \rangle$. $P_i$ locally computes $\langle z \rangle_i = \langle x \rangle_i + \langle y \rangle_i$ ($i \in \{0, 1\}$). Since $\langle z \rangle_0 + \langle z \rangle_1 = \langle x \rangle_0 + \langle x \rangle_1 + \langle y \rangle_0 + \langle y \rangle_1 = x + y$, $(\langle z \rangle_0, \langle z \rangle_1)$ is a correct secret sharing pair of $z$.
- *Scalar-multiplication*: $\langle y \rangle = c \cdot \langle x \rangle$, where $c$ is a public scalar. $P_i$ locally computes $\langle y \rangle_i = c \cdot \langle x \rangle_i$ ($i \in \{0, 1\}$). Obviously, $(\langle y \rangle_0, \langle y \rangle_1)$ is a correct secret sharing pair of $y$.

- *Multiplication*: $\langle z \rangle = \langle x \rangle \cdot \langle y \rangle$ is achieved with the help of a Beaver-triplet $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, where a and b are uniformly random values and $c = a \cdot b$. Then the parties compute $\langle e \rangle = \langle x \rangle + \langle a \rangle$ and $\langle f \rangle = \langle y \rangle + \langle b \rangle$. In this way, $x$ and $y$ are secretly masked by random values. $e$ and $f$ can be reconstructed without leakage. $P_i$ finishes the computation by computing locally $\langle z \rangle_i = -i \cdot e \cdot f + f \cdot \langle x \rangle_i + e \cdot \langle y \rangle_i + \langle c \rangle_i$.
- *Other functions*: *Comparison* protocol is implemented under the Boolean sharing [15], which enables bit decomposition. In this way, the sign of the subtraction is obtained as a comparison result. Non-linear functions are implemented with *Addition* and *Multiplication*, with the help of approximation methods like Taylor series expansion [23, 38].

## 3 ALGORITHM DESIGN

In this section, we discuss the secure relational operators of SQL and reveal our design motivations. We first describe how to implement a fully secure version as the baseline and analyze its bottlenecks. Then, we introduce algorithm optimizations for vertically partitioned collaborative data analysis, which can greatly improve the performance with reasonable security compromises.

### 3.1 Oblivious baseline

We first illustrate the most crucial relational operators with examples. Since we adopt a different Secret Sharing scheme from existing work, tedious algorithm modifications are needed to implement a baseline. Here, we focus on the optimization and adaptation of `SortByKey`, which is the bottleneck of the computation. Finally, we summarize the problems with this baseline.

**Relational Operators.** `JOIN` is implemented with a brute-force approach. Suppose we have two relations R and S with $m$ and $n$ tuples respectively and join them based on a predicate $\theta$: $R.ID = S.ID$, as illustrated in Figure 3. The result is the cartesian product of R and S, and each tuple is augmented with a boolean bit denoting whether the predicate is true. The cardinality of the resulting relation explodes to $(m \cdot n)$.

Next, we provide a simple example and illustrate the steps of `GROUP-BY-AGG` in Figure 4. For simplicity, we assume in the example that the two input relations are aligned. So we skip the `JOIN` step and only consider the `GROUP-BY-AGG` operator.

- *Sort*: sort the relation with secure `SortByKey` protocol based on column $k$ and result in columns $k'$ and $v'$.

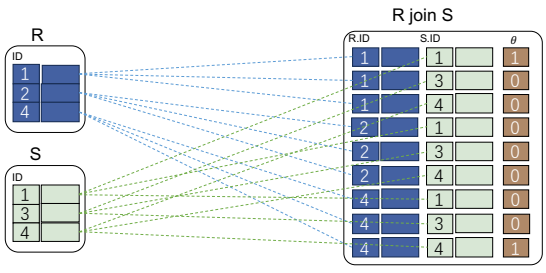


**Figure 3: Illustration of oblivious JOIN**

- *Agg*: aggregate column $v'$ based on the group identifier bit. First, we need to calculate $b$, the binary identifier for the group starting position, based on the group key $k'$. If the group key of the current row is not equal to the previous row, then the new group starts, i.e. $b[i] := k'[i-1] \neq k'[i]$. Next, we calculate the aggregation column *ag* row by row. When encountering the starting point of a group, we reset the aggregation value. Otherwise, we accumulate the current row of $v'$ into the aggregation value, i.e. $ag[i] := b[i] \cdot v'[i] + (1 - b[i]) \cdot (ag[i-1] + v'[i])$. Note that the comparison operators and arithmetic protocols used in these and subsequent formulas are the ones we introduced in section 2.
- *Mask*: mask the unnecessary values. For security reasons, intermediate information other than the output should not be revealed. Therefore, only the last element in each group of accumulated column *ag* should be retained. Similar to the aggregation above, we first calculate the binary identifier $f$ to indicate the end position of a group, i.e. $f := k'[i] \neq k'[i+1]$. Then, we set the unnecessary rows to a predefined invalid number x, i.e. $k'[i] := f[i] \cdot k'[i] + (1 - f[i]) \cdot x$ and $ag[i] := f[i] \cdot ag[i] + (1 - f[i]) \cdot x$.
- *Shuffle*: apply secure `Shuffle` protocol to the relation and reveal the result. If the result is revealed directly without shuffling, we can infer the size of each group from the position of valid rows.

**SortByKey.** From the above description, we know that `Shuffle` and `SortByKey` protocols are crucial for implementing relational operators. The sorting network such as Batcher's sort [2] is a popular solution among existing works [24, 34], due to its input-independent control flow and ease of parallelization. However, it heavily relies on `comparison` protocol, which is inefficient in MPC. Therefore, we implement the stable radix sort instead. Radix sort is implemented based on the ABY3 protocol in existing work [12] and we redesigned it under Additive Secret Sharing for the first time. To better understand radix sort and facilitate our discussion of how it inspires the later optimization of the `GROUP-BY` operator, we first explain the algorithm process on the left side of Figure 5. This example shows the sorting of column K, i.e. the list [1, 0, 3, 2].

First, we decompose the elements in column $K$ into binary digits [01, 00, 11, 10]. The Least Significant Bit (LSB) and the Most Significant Bit (MSB) are denoted as *K_bin_0* and *K_bin_1* respectively. We append column *index* at the end to analyze the destination index of each element after sorting. Then we sort the relation table according to each bit column from LSB to MSB. After that, column $K$ is sorted. We assemble the destination indexes after sorting of original elements into a vector as $p = [1, 0, 3, 2]$, which uniquely identifies the sorting operation and is called a *permutation vector*. For example, 1 is the destination index of the first element in K after sorting. $p(K)$ represents sorting the key K using p.

To reduce the workload of sorting, we do not need to sort the entire relation table each time. Instead, we can process the digits one by one from the LSB. Each time, we use the `GenPerm` function to generate the permutation vector corresponding to the current digit, and then merge it with the previous permutation vector using the `Compose` function. In the end, we can obtain the end-to-end

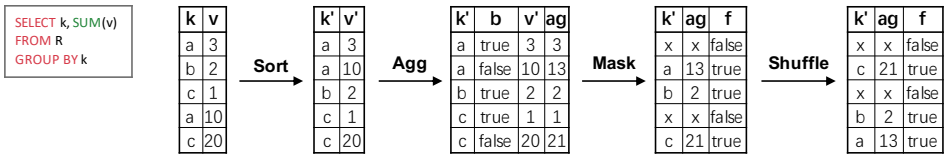


**Figure 4: Illustration of oblivious GROUP-BY-AGG**

permutation vector. Similarly, when implementing `SortByKey`, we first generate the permutation vector based on the key column and then sort all value columns only once.

On the right side of Figure 5, we show the function dependencies in the implementation of `SortByKey` and we have to design these functions based on Additive Secret Sharing. We mainly explain the core implementation path shown by the red arrow. Since the adaptation of the remaining functions to existing work [12] is straightforward, we will not go into detail to save space.

We first implement the `SecurePermutation` and detailed steps are shown on the left side of Figure 6. Although the operator has been implemented in previous work with the same MPC scheme as ours [19], we have made two optimizations for our scenario:

- We remove the offline part. `SecurePermutation` is called by subsequent functions with an input-independent random permutation vector $\pi$, which can be directly sent to the TTP without the need for complex offline services.
- Our protocol is designed for Boolean Sharing inputs. If $X$ is a Boolean share, it can be represented with fewer bits based on its value range, resulting in a shorter random mask $R$ and saving communication when opening $(X - R)$. In our subsequent calls to this function, $X$ is a permutation vector whose bit length depends on the number of samples to be sorted. Only 32 bits are needed to sort more than 4.2 billion data, and communication is then reduced by 50% compared to using 64-bit Additive Secret Sharing.

Now we use `SecurePermutation` to implement the `Shuffle` protocol in the middle of Figure 6. Each of the parties generates a random permutation vector locally, i.e. $\pi_1$ and $\pi_2$, which are used as inputs to call `SecurePermutation` in turn. In this way, the two parties shuffle the input $\langle X \rangle$ collaboratively. Note that each party only holds a single-step permutation vector, which ensures the security of `Shuffle`.

Finally, we represent the implementation of the `SecretSort` protocol on the right side of Figure 6. The function of `SecretSort` is similar to `SecurePermutation`, which sorts the target vector using a given permutation vector, but the difference is that the permutation vector of `SecretSort` is secretly shared. With the help of `Shuffle`, the shared permutation vector $\langle \pi \rangle$ and the target vector $\langle X \rangle$ are shuffled simultaneously, and the shuffled permutation vector $\langle \pi' \rangle$ is then opened. After that, the two parties can use $\pi'$ to permute the local shares of the shuffled target vector $\langle X' \rangle$.

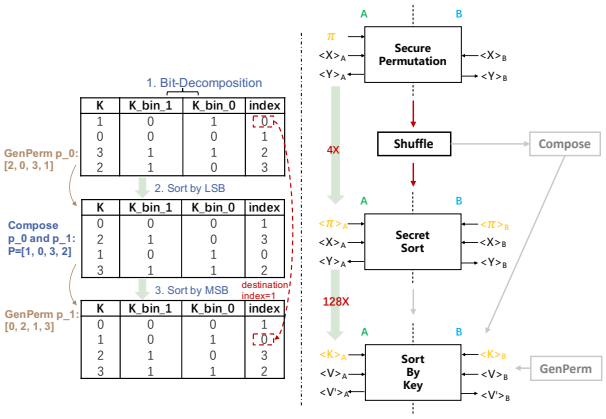


**Figure 5: SortByKey implemented with radix sort**

- Correctness: Since the same shuffling is applied to $\pi$ and $X$, their element-wise correspondence remains the same. Therefore, each element of $\pi'$ still records the target position of each element of $X'$, which is equivalent to the original permutation.
- Security: Although we open $\pi'$, neither party can reverse the shuffle, as we mentioned above. This is similar to adding a random mask to $\pi$, so the masked $\pi'$ can be opened without sacrificing security.

The modification of `GenPerm` is trivial, and the implementation of `Compose` is similar to that of `SecretSort`, so we omit them here. By using these functions, we can finally implement *SortByKey*.

**Bottlenecks.** Although we have implemented the basic version of joint analysis, the performance of this version is impractical due to the following issues:

- After the `JOIN` operator, the size of the result relation rapidly expands, and all subsequent calculations need to be executed on this scale, resulting in too much redundant computation and storage. Therefore, it is difficult to continue when the input scale is larger than ten thousand.
- `GROUP-BY` relies on the time-consuming `SortByKey` protocol. There is a huge time consumption when there are many group-by keys.
- `AGG` requires accumulation row by row, and the result of this row depends on the previous row. Each aggregation requires a secret `Multiplication`, which involves two rounds of communication. Therefore, with the increase in the number of rows, the latency becomes unacceptable. Another problem is that `AGG-MIN` and `AGG-MAX` are inefficient, because they rely on inefficient comparison protocols.

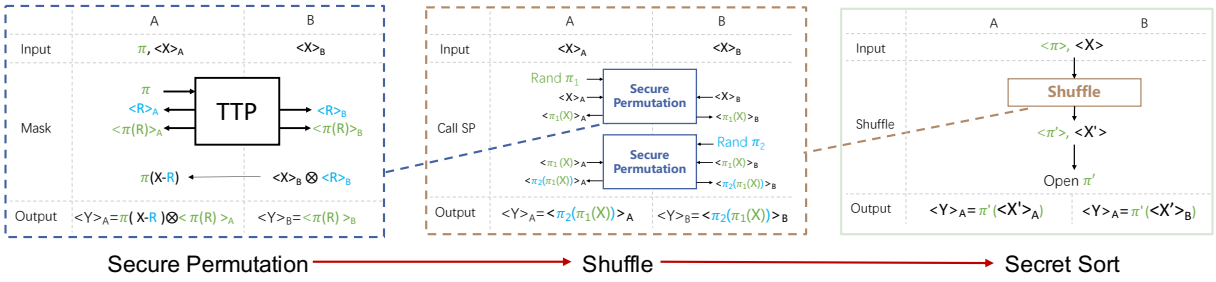


**Figure 6: Implementation of Secret Sort protocol**

## 3.2 Vertical Optimizations

Existing works mainly focus on horizontal scenarios and fully encrypted scenarios. To fill the gap in vertical scenarios, we design appropriate security trade-offs to achieve scalability.

**JOIN.** Many redundant entries are generated by `JOIN`. For example, in the example in Figure 3, only 2 of the 9 entries in the result are valid. If we allow the size of the intersection to be leaked, we can shuffle the data once, and then open the predicate $\theta$, which greatly reduces the number of relations, and reduces the time consumption of subsequent steps. Due to the shuffle operation, it is not possible to restore the original entries corresponding to the intersection result, so only the size of the intersection is leaked. This is the first compromised `JOIN` implementation.

Collaborative modeling in the vertical scenario has been widely applied. It is assumed that the participants have aligned their samples before modeling, which is usually accomplished using Private Set Intersection (PSI) [20, 35]. Similarly, `EQUAL-JOIN` is widely used in collaborative data analysis to align the samples of participants. Therefore, we can provide `PSI-JOIN` as an efficient implementation of `EQUAL-JOIN` for the participants who can tolerate knowing the intersection of samples. We implement ECDH-PSI [27] because it is easy to understand and more communication-friendly compared to other algorithms.

In addition to solving the problem of dimension expansion, the main advantage of implementing `EQUAL-JOIN` with PSI is that the resulting data is private instead of secretly shared. Private data refers to data that exists in plaintext form within its owner and is unknown to other participants. The existence of private data can provide better performance for MPC protocols. For example, As shown in Figure 6, the execution time of `SecretSort` is four times that of `SecurePermutation` simply because the permutation vector of `SecretSort` is a secretly shared $\langle\pi\rangle$, unlike `SecurePermutation` where it is a private $\pi$. In the optimization of `GROUP-BY` later, we further leverage this feature.

**GROUP-BY.** Inspired by the idea of radix sort and the private feature brought by PSI, we propose five optimizations. We first explain them, followed by an example to illustrate their application.

Opt-1: *Sort payload only once.* `GROUP-BY` need to sort the relation table with multiple keys, and the communication cost is proportional to the payload size. Similar to the `SortByKey` protocol, we first generate a permutation vector composed of all the GROUP-BY keys, and then perform a one-time sorting of the entire relation, thereby reducing communication. As in the example later on, there are four GROUP-BY keys, so the communication for sorting the payload is reduced by 75%.

Opt-2: *Merge local keys.* `SortByKey` is a very time-consuming protocol. Notice that there is no priority among the GROUP-BY keys. To utilize the private feature, we first merge multiple local keys to reduce the number of calls to secret sorting protocols. After optimizing the vertical scenario, `SortByKey` is called at most once. As in the example later on, there are four GROUP-BY keys, so the sorting time is reduced by 75%.

Opt-3: *Choose cheaper secret protocol.* When implementing `SortByKey`, it is necessary to process each decomposed binary bit, and the time consumption of processing each bit is more than twice that of `SecretSort`. In SCQL, MPC defaults to using 64-bit numbers, so the time consumption of `SortByKey` is above 128 times that of `SecretSort`. Moreover, the time consumption of the `SecretSort` operator is 4 times that of `SecurePermutation` as we analyzed in section 3.1. The functions of these three protocols are similar, and we should try to use the cheapest one. If we use `SecurePermutation` to replace `SortByKey`, the time for a single sorting is reduced by 500 times.

Opt-4: *Reduce valid bits of keys.* Since `SortByKey` needs to process each binary bit one by one, its complexity is directly related to the number of valid bits in the key column. We can encode the keys with ordinal encoding, which means mapping each unique label to an integer value. The final integer used will not exceed the total number of samples $n$, and the number of valid bits will be at most $\log_2(n)$. For example, for sorting 1 million samples, only 20 bits of valid bits are needed, which reduces the execution time by about 70% compared to the original 64 bits.

Opt-5: *Min/Max aggregation for free.* The radix sorting algorithm gives us an insight that, for multi-key sorting, we should start from the low-priority keys to the high-priority ones. When sorting with the high-priority keys, the ordering of low-priority keys is preserved. For the aggregation functions `MIN` and `MAX`, we can include the related aggregation column as a sorting key with the lowest priority to generate the permutation vector. After sorting, the last element in a group is the desired maximum/minimum value. In this

way, we skip the process of row-by-row aggregation and eliminate the dependency on time-consuming comparison protocols during the process.

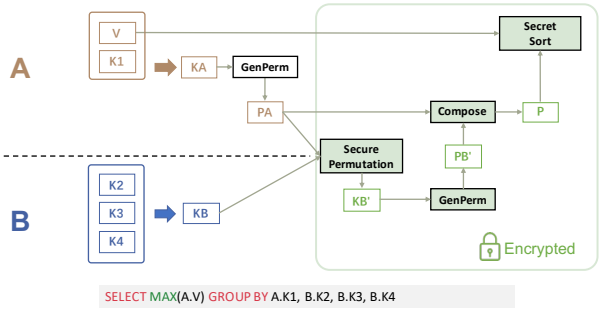


**Figure 7: Vertical GROUP-BY pipeline**

We use a GROUP-BY query with four keys in Figure 7 to illustrate the optimizations. Leveraging Opt-2, we first merge the local keys to obtain $KA$ and $KB$. Since the aggregation function is `MAX`, we use Opt-5 to let aggregation column $V$ participate in this key merging process. $KA$ and $KB$ are generated using Opt-4, and the keys are combined with ordinal encoding. Since $KA$ contains $V$ with the lowest priority, our permutation vector generation should start from $KA$. We use the plaintext `GenPerm` function to generate the corresponding permutation vector $PA$ for $KA$. According to Opt-3, we sort $KB$ based on $PA$ with `SecurePermutation` instead of `SortByKey`. This step involves the data of both participants, so all subsequent processes are encrypted by Secret Sharing. The permutation vector $PB'$ corresponding to the sorted $KB'$ is generated using the secret `GenPerm`, and merged with the $PA$ to generate the end-to-end permutation vector $P$. Finally, we only need to use $P$ to sort the aggregation column $V$ once, corresponding to Opt-1.

**AGG.** To avoid the row-by-row aggregation process, we first adopt the method in [24]. They innovatively employ the Brent-Kung network to achieve a traversing function. Although the communication volume is doubled, the number of communication rounds decreases from $n$ to $O(\log n)$. This optimization makes the performance of the aggregation function practical.

If users allow for some security concessions, the cost of the aggregation function is further reduced to almost zero. Reviewing the aggregation steps in Figure 4, the aggregation formula is: $ag[i] := b[i] \cdot v'[i] + (1 - b[i]) \cdot (ag[i-1] + v'[i])$. If we open the group identifier $b$, `Multiplication` is replaced by `Scalar-multiplication` as described in section 2, which is a completely local computation without interaction. So the time consumption will be almost free. In terms of security, open $b$ is equivalent to revealing the number of samples contained in each group. Taking Figure 4 as an example, it is equivalent to the query: `select count(v) from R group by k`.

## 4 IMPLEMENTATION

In this section, we elaborate on platform design. Firstly, we present the high-level architecture and explain the roles of the core modules. Then, we provide a query example to demonstrate the execution process, which showcases the interaction between the components and highlights the user interface's usability. Further, we introduce a specific annotation, i.e. Column Control List (CCL), that describes users' security requirements.

### 4.1 Architecture

We demonstrate the architecture of SCQL in Figure 8. Though we illustrate SCQL with two participants, Additive Secret Sharing naturally supports three or more participants. Party 1 and Party 2 represent two participants in collaborative data analysis, who are the data owners and also responsible for actual computation. The original data of each participant is stored in its local database. Additionally, we need a trusted third party to deploy centralized service modules. Note that the trusted third party has no access to actual data and is only involved in assisting with the generation of execution plans. We explain the roles of the four core modules in the following:

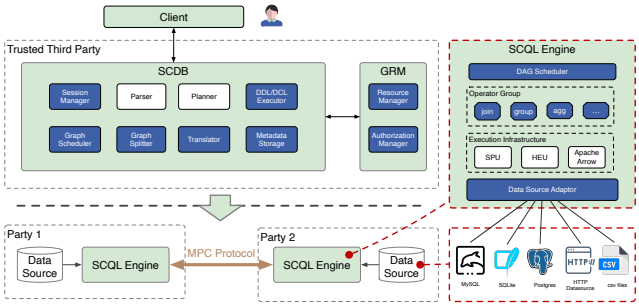


**Figure 8: Architecture of SCQL**

**Client.** It provides users with an interface that allows them to submit queries via the command line and receive execution results.

**GRM.** Global Resource Manager (GRM) supplies SCDB with the necessary information for query execution. GRM plays a role similar to DNS, providing SCDB with services to map logical tables to related physical tables.

**SCDB.** Secure Collaborative DataBase (SCDB) plays the role of a database management system in collaborative data analysis. In order to free users from concerns about the physical distribution of data tables, SCDB creates virtual databases. Users only need to write logical table names in the same way as in the centralized queries. Moreover, the computational tasks are automatically dispatched to the corresponding SCQLEngines, and users do not need to specify the physical address in the query.

SCDB needs a database to store the table schemas and annotations. The details of internal components are better illustrated by concrete workflows, so we leave it in section 4.2.

**SCQLEngine.** It is deployed at each participant and constitutes the execution engine with other participants for collaborative data analysis. It consists of four layers from the bottom up:

- Data Source Adaptor: It connects the local databases and the execution infrastructures. Various database management systems are supported in SCQL.
- Execution Infrastructure: SCQLEngine is a hybrid MPC-plaintext execution engine. It integrates open-source MPC

infrastructures [36][1] to enable joint computation while ensuring data security. It also integrates open-sourced plaintext execution infrastructure[2] for local computations. In different execution steps, we can switch to different infrastructures to meet the security requirements and obtain better performances.

- Operator Group: each node in the execution graph is a SQL operator as we described in section 3. These operators can be divided into two categories in terms of the execution mode. One type is plaintext operators, which delegate the computation to the local database or the plaintext execution infrastructures. The other type is secure operators that are implemented with the help of MPC infrastructures and require coordination with other SCQLEngines from different participants.
- DAG scheduler: It receives the execution graph from SCDB and schedules the execution of nodes.

## 4.2 Workflow

To better understand the user interface and internal components of SCDB, we display the workflow with the case in Figure 2. Here, we assume that SCDB, GRM, and SCQLEngines have been deployed. The data tables have been stored in the local databases and registered in GRM.

**Data Preparation.** The first step is to create a virtual database and two accounts for collaborative data analysis. Similar to a centralized database, there is an initial account named root in SCDB. We complete this step with the root account in Figure 9. `CREATE DATABASES` is used to create a virtual database named *demo*. `SHOW DATABASES` lists all the virtual databases. Then we use `CREATE USER` to create an account named *alice* with the password "alice123". We need to grant permissions to *alice* by using the `GRANT` command, which allows this account to create tables, edit column annotations, and delete tables in *demo*. Similarly, an account for *bob* is created.

```
1  root>   CREATE DATABASES demo
2  root>   SHOW DATABASES;
3  [fetch]
4  1 rows in set: (2.945772ms)
5  +----------+
6  | Database |
7  +----------+
8  | demo     |
9  +----------+
10 root> CREATE USER alice PARTY_CODE "alice" IDENTIFIED BY "alice123"
11 root> GRANT CREATE, GRANT OPTION, DROP ON demo.* TO alice
12 root> CREATE USER bob PARTY_CODE "bob" IDENTIFIED BY "bob123"
13 root> GRANT CREATE, GRANT OPTION, DROP ON demo.* TO bob
```

**Figure 9: Preparation commands at root**

The second step is to add the logical table to the virtual database after switching to the accounts created for data owners in the previous step. We take *alice* as an example and list the commands in Figure 10. The `CREATE TABLE` statement at line 1 adds the table *ta* to the virtual database *demo*, where *TID* is the table identifier registered from GRM. The `DESCRIBE` statement displays the table schema. Moreover, the data owners need to set the CCL with the `GRANT` command for all the columns used. A query must pass the CCL check before execution to ensure that all calculations are in line with the expectations of the data owners. We skip over the explanations of these CCL annotations for now and leave them to be discussed in Section 4.3. Then, *bob* sets up table *tb* in the same way.

```
1  alice> CREATE TABLE demo.ta TID="tid0"
2  alice> DESCRIBE demo.ta
3  [fetch]
4  4 rows in set: (2.581103ms)
5  +--------------+--------+
6  |    Field     |  Type  |
7  +--------------+--------+
8  | id           | string |
9  | credit_rank  | int    |
10 | income       | int    |
11 | age          | int    |
12 +--------------+--------+
13 alice> GRANT ta.ID ON demo TO bob WITH block (Project, Select);
14 alice> GRANT ta.ID ON demo TO bob WITH relax (reveal_join_result);
15 alice> GRANT ta.credit_rank ON demo TO bob WITH must (AsGroupKey);
16 alice> GRANT ta.income ON demo TO bob WITH must (Avg);
```

**Figure 10: Preparation commands at alice**

```
1  alice> use demo
2  [demo]alice> SELECT ta.credit_rank, tb.is_active, COUNT(*) as user_cnt, ...
3  [fetch]
4  12 rows in set: (1.151690583s)
5  +-------------+-----------+----------+------------+------------+
6  | credit_rank | is_active | user_cnt | avg_income | max_amount |
7  +-------------+-----------+----------+------------+------------+
8  |      5      |     0     |    37    | 336016.22  |    7743    |
9  |      5      |     1     |    63    | 18069.775  |    5499    |
10 |                        ...                                   |
11 +-------------+-----------+----------+------------+------------+
```

**Figure 11: Query submission**

**Query Execution.** Now, Alice can submit the query to SCDB, as shown in Figure 11, and get the result. To better understand the execution process of SQL and the internal components of SCDB, we draw the pipeline in Figure 12.

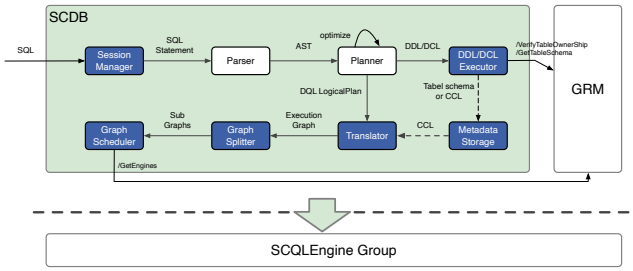


**Figure 12: Query execution pipeline**

After the Client submits a SQL statement, SCDB creates a session for the request and performs user authentication to reject illegal user access. The Parser parses the SQL statement into an Abstract Syntax Tree (AST). Then, the Planner generates a logical plan based

[1] https://github.com/secretflow/spu.
[2] ApacheArrow: https://github.com/apache/arrow.

on the syntax tree. For the DCL (GRANT/REVOKE) and DDL (CREATE/DROP) queries in the data preparation stage, the logical plans are handed over to the DDL/DCL Executor. If it is a DCL plan, the executor sends the permission information to the Metadata Storage. As for the DDL plan, SCDB sends the table ID and user tokens to GRM. After confirming that the user has been granted the table ownership, GRM returns the table schema to SCDB, which is then sent to the Metadata Storage. In this way, a logical table is successfully registered in the virtual database.

For a DQL (SELECT) statement, a logical plan is also generated first, followed by a plan optimization step. We adopt former optimizations in collaborative data analysis [3, 34, 55]. For example, predicate push-down optimization can be used here to reduce the amount of data entering the MPC environment, thus improving the performance. In fact, we reuse the Parser and Planner of the open-source database[3] and make corresponding adaptations for our scenario. For example, we make appropriate modifications to the functionality of Parser according to the syntax we support and select plan optimization configurations in Planner that are suitable for secure collaborative data analysis. Then, the Translator reads the CCL annotations from the Metadata Storage and verifies whether the query complies with the restrictions of the data owners. After the CCL check, the Translator converts the logical plan into an execution graph composed of SQL operators. Next, Graph Scheduler optimizes the execution graph, e.g. merging nodes and removing duplicate nodes, and distributes the split subgraphs to SCQLEngine Group for execution. Finally, the results are returned to the Client via SCDB after the execution is completed.

## 4.3 Column Control List

CCL is proposed to describe users' security requirements. Each CCL configuration can be described by a triplet: <`src_column`, `dest_party`, `constraint`>, which indicates that `src_column` is accessible to `dest_party` with the `constraint`. The syntax for setting CCL is: `GRANT src_column TO dest_party WITH constraint`.

There are three types of constraints:

- *mustOpSet*. The `src_column` must go through specified operations before revealing. For example, if we define $must = \{sum|avg, union\}$, it means that the `src_column` must be horizontally unioned with other participants and aggregated by *sum* or *avg* function before being output to the `dest_party`. The list of operators that we can configure is displayed in Figure 13.

| 1. Relational operations | 2. Expression operations |
|---|---|
| – Project<br>– Select<br>– Join<br>  + AsJoinKey<br>  + AsJoinPayload<br>– GroupBy<br>  + AsGroupKey<br>  + AsGroupPayload<br>– Union | – AggregationFunc<br>  + Sum, Count, Avg, Min, Max<br>– ArithmeticOp<br>  + Add, Sub, Mul, Div<br>– ComparisonOp<br>  + Equal, NE, LT, LE, GT, GE<br>– LogicalOp<br>  + And, Or, Not |

**Figure 13: Operator list**

[3]TiDB: https://github.com/pingcap/tidb

- *blockOpSet*. The listed operators are prohibited from being performed on `src_column` during the execution. If a column is not configured, we add a default configuration to it, i.e. $block = \{all\}$, which indicates that no operations can be performed on this column.
- *relaxRule*. It is used to define intermediate information that is allowed to be opened, thus improving the performance of the corresponding operator. Currently, SCQL supports the following four rules:
  - *public*: `src_column` can be reveal to `dest_party`. When the operands are configured as public, we invoke plaintext relational operators in the execution graph, the same as the optimizations in SMCQL and Conclave.
  - *reveal_join_size*: when `src_column` is used as a JOIN condition, the intersection size can be revealed to designated `dest_party`. In this case, we adopt the first compromised `JOIN` mentioned in section 3.2.
  - *reveal_join_result*: when `src_column` is used as a JOIN condition, the intersection can be revealed to `dest_party`. In this case, we adopt the `PSI-JOIN` in section 3.2.
  - *reveal_group_size*: when `src_column` is used as a GROUP-BY key, The sample size of each group can be revealed to `dest_party`. In this case, we adopt the compromised `AGG` in section 3.2.

In summary, *mustOpSet* and *blockOpSet* are used for static semantic checking to reject illegal queries. *relaxRule* is used for runtime optimization to reduce secret computations and speed up execution.

Now, we are ready to review the CCL annotations of alice in Figure 10 and analyze the effect of each configuration. According to the configuration in line 13, projecting on $ta.ID$ and its derived columns or using $ta.ID$ as a selection predicate is prohibited. If Bob submits the following three queries, they will be rejected:

- `select id from ta;`
- `select count(id) from ta;`
- `select income from ta where id = 'xxx';`

Next, if both Alice and Bob configure the *relaxRule* for their own column *ID* as in line 14, the execution of `JOIN` will invoke the efficient `PSI-JOIN` in section 3.2. The configurations in lines 15 and 16 describe the usage of columns $ta.credit\_rank$ and $ta.income$ in the target query. During the configuration process, we should try to minimize the authorized scope as much as possible to prevent data abuse. If we want to expand their applications, we only need to modify the configuration accordingly. For example, if we not only want to calculate the average income but also want to calculate the maximum income, we only need to change the configuration in line 16 and set $must = \{avg|max\}$.

# 5 EXPERIMENTS

In this section, we conduct comprehensive experiments to validate the performance of SCQL. In the building block test, we test the proposed MPC protocols in section 3.1. Then, we compare the performance of different implementations of relational operations in section 3.2. Finally, we analyze the effectiveness of our optimization in end-to-end queries and demonstrate the scalability of SCQL.

## 5.1 Experimental setup

**Settings.** We set the ring size of the secret sharing schema as 64 bits and 20 bits are used for the fractional part. Zero sharing technique is applied to reduce the interaction cost with the help of pseudo-random functions (PRF) [37]. Taking the *SecurePermutation* protocol in Figure 6 as an example, party A generates a random vector as $\langle R\rangle_A$ and party B generates two random vectors as $\langle R\rangle_B$ and $\langle \pi(R)\rangle_B$. Simultaneously, the TTP generates the same three random vectors by using the PRF negotiated with party A and party B. TTP calculates the correction item, i.e. $\langle \pi(R)\rangle_A = \pi(\langle R\rangle_A + \langle R\rangle_B) - \langle \pi(R)\rangle_B$ and sends it to A. In this way, the communication volume for generating correlated random vectors is greatly reduced. Moreover, we implement ECDH-PSI with the Curve25519.

**Environment.** We conduct experiments on two Alibaba Cloud servers with 16 vCPU and 128GB RAM each. The CPU model is Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz. The network condition is manipulated with the Linux traffic control tool (e.g., *tc* command). To be specific, we consider the transmission *bandwidth* and *latency*. In the WAN setting, we limit the network connection by 100 Mbps with 20 ms of roundtrip latency.

## 5.2 Evaluation of Secure Sorting

In this section, we evaluate our redesigned `SortByKey` protocol, which is the bottleneck of `GROUP-BY`. We first test *SortByKey* protocol and the related protocols as described in section 3.1. Then we compare our implementation with the commonly used open-source implementation of EMP[4], which is based on Bitonic Sort.

**Related protocols.** We adopt the WAN setting to evaluate these communication-dominated protocols and vary the input size to test the execution time. The results are shown in Table 1.

**Table 1: Execution time (ms) of SortByKey**

| Input | SP0 | SP1 | SecSort | SBK | SBK_valid |
|---|---|---|---|---|---|
| $10^3$ | 10.1 | 20.3 | 50.7 | 3404 | 643 |
| $10^4$ | 10.4 | 20.9 | 57.3 | 7960 | 2076 |
| $10^5$ | 67.3 | 136 | 536 | 75566 | 23007 |
| $10^6$ | 691 | 1402 | 5367 | 769886 | 267668 |

Now we analyze whether the time ratios of the three protocols are consistent with our previous complexity analysis. Here, SP refers to the `SecurePermutation` protocol, and SP0 and SP1 represent that the input permutation vectors are located in the two participants P0 and P1 respectively. SecSort indicates the *SecretSort* protocol and SBK means *SortByKey* protocol.

When the input size exceeds $10^5$ and communication dominates the execution time, the result is consistent with theoretical value: (1) SP1 is approximately twice the time of SP0, because we used Zero Sharing technology in section 5.1 and we assume that TTP and P0 are in the same local area network. (2) The execution time of SecSort is supposed to be 7 times that of SP0, due to the fact that *SecretSort* involves two rounds of SP0, two rounds of SP1, and one round of open masked values. (3) The execution time of SBK is about 140 times that of SecSort, which is consistent with our analysis in section 3.2.

We also test Opt-4 in section 3.2, i.e. reduce valid bits of keys, and the results are in the column SBK_valid. The protocol automatically determines the number of valid bits based on the input size. The smaller the input size is, the fewer valid bits we get, thus resulting in higher optimization ratios. Here, we achieved a performance improvement of 1.9 to 4.3 times.

**Comparison with EMP.** We demonstrate the results in Figure 14, with execution time plotted in log-scale. Since the asymptotic complexity of Bitonic Sort is $O(nlog^2n)$, while our Radix Sort is $O(nl)$, our advantage becomes more obvious as the input size increases. In the end, we sorted one million 64-bit data in only 4.5 minutes, while EMP required 9.4 hours, which is approximately 126 times slower than our implementation.

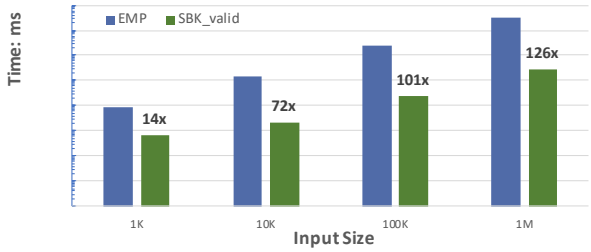


**Figure 14: Compare SortByKey with EMP**

## 5.3 Relational Operation Test

Further, we test the performance of three core relational operations introduced in section 3.1 and section 3.2. We vary the input size and compare the performance of different implementations for `JOIN`, `AGG`, and `GROUP-BY`.

**JOIN.** We have three implementations of `JOIN`. Plaintext-JOIN is used in previous work [3, 55] when the join column is public. Oblivious-JOIN and PSI-JOIN are as we introduced in sections 3.1 and 3.2. PSI protocol is computation-dominated, so we test it under the LAN setting without network restrictions and the results are in Table 2. We can see that: (1) Oblivious-JOIN is difficult to scale, as the memory runs out with an input size of only 10,000. (2) PSI-JOIN and Plaintext-JOIN both have low execution time even with millions of data, but PSI-JOIN is 10 times slower than Plaintext-JOIN under this amount of data.

**Table 2: Performance of JOIN (Time: s)**

| Input Size | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
|---|---|---|---|---|---|
| **Plaintext-JOIN** | 0.001 | 0.011 | 0.121 | 2.602 | 30.701 |
| **PSI-JOIN** | 0.040 | 0.423 | 3.291 | 33.284 | 309.401 |
| **Oblivious-JOIN** | 10.400 | OOM | OOM | OOM | OOM |

**AGG.** We test two implementations of the `AGG` as described in section 3.2 and the results are in Table 3. The prefix `Plain` represents

[4]https://github.com/emp-toolkit/emp-sh2pc

the version with secure concession, which reveals the group identifier. The prefix Obliv represents the version that uses the Brent-Kung network for acceleration. Furthermore, the performance of AGG also varies depending on whether a comparison protocol is needed. Therefore, we chose Sum and MAX as the representatives for this performance testing.

**Table 3: Performance of AGG (Time: s)**

| Input Size | | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
|---|---|---|---|---|---|---|
| LAN | **Obliv-Sum** | 0.011 | 0.031 | 0.174 | 1.971 | 20.012 |
| | **Obliv-Max** | 0.062 | 0.094 | 0.383 | 3.824 | 39.201 |
| | **Plain-Sum** | 0.001 | 0.001 | 0.011 | 0.102 | 0.932 |
| | **Plain-Max** | 0.001 | 0.001 | 0.001 | 0.008 | 0.076 |
| WAN | **Obliv-Sum** | 0.512 | 0.773 | 2.931 | 17.630 | 138.255 |
| | **Obliv-Max** | 2.291 | 3.370 | 8.494 | 41.891 | 301.673 |
| | **Plain-Sum** | 0.001 | 0.001 | 0.012 | 0.110 | 0.932 |
| | **Plain-Max** | 0.001 | 0.001 | 0.001 | 0.008 | 0.071 |

We observe that: (1) As there are no network limitations in the LAN setting, the test results represent the computational cost. The result gap between the WAN and LAN settings represents the communication cost. Comparing these two costs, we found that 85.5% to 97.9% of the time for Oblivious-AGG is spent on network communication. (2) The WAN setting is more realistic for showcasing the true effect of the Oblivious operator in real-world applications. The results in WAN setting the runtime of Obliv-MAX is 2.2 to 4.5 times that of Obliv-SUM, due to the dependence on the comparison protocol. (3) Although the oblivious operators are more than 100 times slower than plaintext ones, the slowest Obliv-MAX only takes 302 seconds, which is practical enough. (4) Since the group identifier is revealed, the aggregation processes for Plain-Sum and Plain-Max are both local computations, therefore their runtime is almost the same.

**GROUP-BY.** We test the performance of the three implementations of GROUP-BY, summarized in Table 4. In this test, we assume that each of the two parties holds one key. Plaintext GROUP-BY (P-GB) refers to the case where the GROUP-BY key is public and can be sorted in plaintext. Oblivious GROUP-BY (O-GB) calls the SortByKey once for each key. Vertical GROUP-BY (V-GB) includes all the optimization points mentioned in section 3.2.

**Table 4: Performance of GROUP-BY (Time: s)**

| Input Size | | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
|---|---|---|---|---|---|---|
| LAN | **P-GB** | 0.003 | 0.030 | 0.263 | 2.640 | 23.252 |
| | **V-GB** | 0.071 | 0.684 | 4.401 | 53.636 | 518.810 |
| | **O-GB** | 0.262 | 1.442 | 15.932 | 182.207 | 1639.573 |
| WAN | **P-GB** | 0.021 | 0.053 | 0.291 | 2.716 | 23.846 |
| | **V-GB** | 1.812 | 6.360 | 48.563 | 422.304 | 4049.305 |
| | **O-GB** | 8.720 | 27.982 | 248.827 | 1804.261 | 15979.334 |

From the table, we can see that: (1) Similar to the analysis of oblivious AGG, we learn that the runtime of O-GB and V-GB is dominated by communication. In the WAN setting, communication accounts for approximately 87.2% to 97.0% of the execution time. (2) Through optimization in the vertical scenario, the execution time in O-GB is about 3 to 4 times that in V-GB. (3) If the number of keys in GROUP-BY continues to increase, the runtime of oblivious GROUP-BY will continue to increase linearly, while the runtime of vertical GROUP-BY will remain almost unchanged, further expanding its advantage. (4) When the input size is 10 million , O-GB requires 4.4 hours of execution time, while V-GB only requires 1.1 hours. Even with significant optimization, sorting operations on large datasets remain time-consuming, making GROUP-BY the main bottleneck.

## 5.4 End-to-end test

In this section, we first use a case study to demonstrate how the various optimizations improve execution performance step by step. Then, we test the scalability of SCQL in horizontal and vertical scenarios respectively.

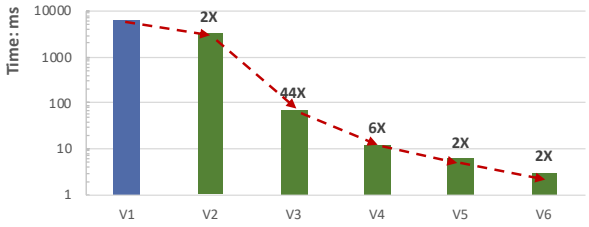


**Figure 15: Case study of optimizations**

**Optimization Test.** We use the query in Figure 2 to perform end-to-end performance tests in the WAN setting in order to show the effect of each optimization vividly. We first train a baseline model, and then gradually add a new optimization on top of the previous one. We show the running time of each version and the improvement factor relative to the previous version in Figure 15, with the running time plotted in the log scale. The description of the six versions is as follows:

- V1: baseline, implemented with oblivious JOIN, oblivious GROUP-BY, and Oblivious AGG. In this version, we rewrite the Secrecy [34] using the Additive Secret Sharing and adopted the optimization of Scape [24] for AGG.
- V2: integrating logical plan optimizations, such as predicate push-down and execution graph node merging.
- V3: allowing to reveal the intersection size.
- V4: allowing to reveal the intersection and replacing the oblivious JOIN with PSI-JOIN.
- V5: replacing oblivious GROUP-BY with vertical GROUP-BY.
- V6: allowing to reveal the group sizes and replacing the oblivious AGG with plaintext AGG

Since oblivious JOIN cannot scale, the input size of each party is 2000 in this experiment and the intersection size is 1000. Our result analysis is as follows: (1) V1->V2, predicate push-down has a significant effect, reducing the data entering the secure computation by half, resulting in a doubling of performance. (2) V2->V3: according to the description in section 3.2, shuffling the relation once is required before revealing the intersection predicate, and

this secure operation increases the cost of JOIN. However, after the predicate is revealed, the input size of the subsequent GROUP-BY and AGG operations is greatly reduced, resulting in a significant improvement of 44 times. (3) V3->V4: Since both GROUP-BY and AGG in V3 are performed with a small input size, the JOIN operator becomes the bottleneck. In V4, the oblivious JOIN is replaced with a lightweight PSI-JOIN, resulting in a further 6-fold improvement in performance. (4) V4->V5: the 2-fold improvement in this step is mainly due to the decrease in the number of `SortByKey` calls. And the input size is small, allowing a smaller valid bit to be used for GROUP-BY keys. (5) V5->V6: the final 2-fold improvement comes from using plaintext AGG. In summary, the end-to-end time consumption of this typical vertical scenario query is reduced by 1991 times from the rewritten Secrecy (Additive Secret Sharing) to the best version.

**Scalability Test.** We select representative queries from the TPC-H benchmark and transform them into test cases for collaborative data analysis. There are five queries in this test.

- Q1:from TPC-H Q6, the original query is a single-table query to compute global aggregate functions. We make both participants hold a local table with this schema for collaborative data analysis. Since their tables have the same schema, this is a typical horizontal scenario. We re-write the original query to union the tables before executing the global aggregation.
- Q2: from TPC-H Q1, the original query is a single-table query for computing group aggregate functions. As before, we transformed it into a horizontal scenario for testing.
- Q3: from TPC-H Q14, the original query is a two-table join query for computing global aggregate functions. We make each participant hold one of the original tables, thus transforming it into a test case for the vertical scenario.
- Q4: from TPC-H Q12, the original query is a two-table join query for computing group aggregate functions. We followed the same process as before to transform it into a vertical scenario.
- Q5: since the GROUP-BY clause in the previous query has only one key, we modify TPC-H Q1 to simulate a vertical scenario where both participants hold a key. We vertically partitioned the original table and distributed it to the two participants, with one side holding a GROUP-BY key.

We test in the WAN setting and report the runtime in Table 5. This table tells us: (1) Q1 and Q2, the two test cases in horizontal scenarios, have short execution times because of the optimization technique of pushing down the aggregation functions to the local side. This technique significantly reduces the amount of data that actually needs to be computed secretly, therefore they can easily handle massive data up to tens of millions. (2) Q3 and Q4 have much shorter execution times than Q5 because they do not require the time-consuming `SortByKey` operation. Q4 is slightly more complex than Q3 because it includes a `SecurePermutation`, but overall their execution times are similar because Q3 contains a time-consuming EQUAL operator in its expression. (3) Q5 represents the most complex scenario when using vertical optimization because it requires the `SortByKey`. And the running time for Q5 is approximately 10 times longer than that of Q3 and Q4. (4) For massive data in the tens of millions, simple queries can be completed in under 10 minutes, and even the most complex query, Q5, can be completed in about one hour. This shows that SCQL has excellent scalability.

**Table 5: Scalability test on five typical queries (Time: s)**

| Query | | Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|---|---|
| LAN | $10^3$ | 0.02 | 0.19 | 3.53 | 0.09 | 0.17 |
| | $10^4$ | 0.02 | 0.21 | 3.80 | 0.39 | 0.74 |
| | $10^5$ | 0.07 | 0.38 | 6.88 | 3.38 | 6.60 |
| | $10^6$ | 0.63 | 1.94 | 37.75 | 32.79 | 72.61 |
| | $10^7$ | 6.40 | 18.20 | 347.31 | 330.16 | 768.25 |
| WAN | $10^3$ | 0.30 | 8.85 | 4.32 | 1.08 | 3.44 |
| | $10^4$ | 0.30 | 8.87 | 4.69 | 1.44 | 7.38 |
| | $10^5$ | 0.36 | 9.05 | 8.28 | 4.79 | 45.68 |
| | $10^6$ | 0.89 | 10.60 | 42.34 | 36.17 | 364.04 |
| | $10^7$ | 6.70 | 26.84 | 384.17 | 356.85 | 3926.51 |

## 6 RELATED WORK

**Outsourced Data Analysis.** The data owners encrypt the data and send it to external nodes to perform outsourcing computations. There are three lines of work in this area: work based on TEE [14, 17, 48, 54, 57], work based on MPC [24, 26, 34], and work based on encryption techniques [42, 44, 45, 47, 49, 52]. In practice, sensitive data is often not allowed to leave its source, so we do not focus on the outsourcing mode.

**DP-Based Data Analysis.** Differential privacy (DP) is widely used to protect the security of the results [25, 39], and recent works [4, 5] have combined DP with MPC to improve performance through controlled security relaxation. However, data analysis in real-world applications often requires sufficient accuracy, so our system does not integrate DP methods.

## 7 CONCLUSION

This work proposed SCQL, a framework that enables mutually distrusting parties to make queries of their collective data. SCQL provides data analysts with a user-friendly API to support general-purpose queries and automatically translates these queries into MPC to ensure security. We design more efficient underlying MPC protocols as well as relational operations in SQL to improve efficiency and scalability. Comprehensive experiments demonstrate the effectiveness of our optimizations. In the future, we are interested in integrating malicious protocols with heightened security levels and enhancing the performance of hybrid-scenario queries.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and program chairs for their assistance in revising the paper. We are also grateful for the contributions of Benyu Zhang, Jun Qi, Teng Teng, and Kai Liu in the early stages of this work, as well as Haoqi Wu for his efforts on the final version of the paper. Lastly, we thank all members of the SecretFlow team for their support throughout this project.

# REFERENCES


[1] [n.d.]. General Data Protection Regulation (GDPR). https://gdpr-info.eu/. Accessed April 4, 2010.

[2] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[3] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *Proc. VLDB Endow.* 10, 6 (2017), 673–684.

[4] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[5] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[6] Donald Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*. Springer, 420–432.

[7] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and Efficiently Searchable Encryption. In *Advances in Cryptology - CRYPTO 2007*, Alfred Menezes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 535–552.

[8] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. *Order-Preserving Symmetric Encryption*. 224–241. https://doi.org/10.1007/978-3-642-01001-9_13

[9] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. *Order-preserving encryption revisited: improved security analysis and alternative solutions*. 578–595. https://doi.org/10.1007/978-3-642-22792-9_33

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[11] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2810103.2813700

[12] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An efficient secure three-party sorting protocol with an honest majority. *Cryptology ePrint Archive* (2019).

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security* (Nov 2011), 895–934. https://doi.org/10.3233/jcs-2011-0426

[14] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2020. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.

[15] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[16] Daniel Escudero. 2022. An introduction to secret-sharing-based secure multiparty computation. *Cryptology ePrint Archive* (2022).

[17] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (oct 2019), 169–183. https://doi.org/10.14778/3364324.3364331

[18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. https://doi.org/10.1561/9781680835090

[19] Wenjing Fang, Derun Zhao, Jin Tan, Chaochao Chen, Chaofan Yu, Li Wang, Lei Wang, Jun Zhou, and Benyu Zhang. 2021. Large-scale secure XGB for vertical federated learning. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 443–452.

[20] Fangcheng Fu, Huanran Xue, Yong Cheng, Yangyu Tao, and Bin Cui. 2022. Blindfl: Vertical federated machine learning without peeking into your data. In *Proceedings of the 2022 International Conference on Management of Data*. 1316–1330.

[21] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[22] Oded Goldreich. 2003. Cryptography and cryptographic protocols. *Distributed Computing* (Sep 2003), 177–199. https://doi.org/10.1007/s00446-002-0077-1

[23] Robert E Goldschmidt. 1964. *Applications of division by convergence*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[24] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1740–1753.

[25] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. 2017. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1389–1406.

[26] Zhian He, Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, Siu Ming Yiu, and Eric Lo. 2015. SDB: a secure query processing system with data interoperability. *Proceedings of the VLDB Endowment* (Aug 2015), 1876–1879. https://doi.org/10.14778/2824032.2824090

[27] Bernardo A Huberman, Matt Franklin, and Tad Hogg. 1999. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM conference on Electronic commerce*. 78–86.

[28] Mihaela Ion, Benjamin Kreuter, Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. 2020. Private Intersection-Sum Protocols with Applications to Attributing Aggregate Ad Conversions. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 370–389. https://eprint.iacr.org/2019/723.pdf

[29] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. 2013. On the power of correlated randomness in secure computation. In *Theory of Cryptography Conference*. Springer, 600–620.

[30] MohammadSaiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. *Network and Distributed System Security Symposium,Network and Distributed System Security Symposium* (Jan 2012).

[31] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. 2013. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics* 29, 7 (2013), 886–893.

[32] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2976749.2978386

[33] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. 557–574.

[34] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. {SECRECY}: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[35] Linpeng Lu and Ning Ding. 2020. Multi-party private set intersection in vertical federated learning. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 707–714.

[36] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. [n.d.]. SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning. ([n. d.]).

[37] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.

[38] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38.

[39] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially Private Join Queries over Distributed Databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 149–162. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/narayan

[40] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2810103.2813651

[41] Pascal Paillier. 2007. *Public-key cryptosystems based on composite degree residuosity classes*. 223–238. https://doi.org/10.1007/3-540-48910-x_16

[42] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 587–602. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/papadimitriou

[43] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. 2017. DStress: Efficient differentially private computations on distributed data. In *Proceedings of the Twelfth European Conference on Computer Systems*. 560–574.

[44] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *2014 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/sp.2014.30

[45] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database Using Semantically Secure Encryption. 12, 11 (jul 2019), 1664–1678. https://doi.org/10.14778/3342263.3342641

[46] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: a {Maliciously-Secure}{MPC} platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*. 2129–2146.

[47] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 85–100.

[48] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[49] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proceedings of the VLDB Endowment* 16, 4 (2022), 601–614.

[50] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. 2019. Secure multiparty PageRank algorithm for collaborative fraud detection. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer, 605–623.

[51] Dawn Xiaoding Song, D. Wagner, and A. Perrig. 2002. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. SP 2000.* https://doi.org/10.1109/secpri.2000.848445

[52] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. (2013).

[53] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[54] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.

[55] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[56] Matei Zaharia, Mosharaf Chowdhury, MichaelJ. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *IEEE International Conference on Cloud Computing Technology and Science,IEEE International Conference on Cloud Computing Technology and Science* (Jun 2010).

[57] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.