Secure Statistical Analysis on Multiple Datasets: Join and Group-By

Gilad Asharov
Bar-Ilan University
Ramat Gan, Israel
Gilad.Asharov@biu.ac.il

Koki Hamada
NTT Corporation
Tokyo, Japan
koki.hamada@ntt.com

Dai Ikarashi
NTT Corporation
Tokyo, Japan
dai.ikarashi@ntt.com

Ryo Kikuchi
NTT Corporation
Tokyo, Japan
9h358j30qe@gmail.com

Ariel Nof
Bar-Ilan University
Ramat Gan, Israel
ariel.nof@biu.ac.il

Benny Pinkas
Bar-Ilan University
Ramat Gan, Israel
Aptos Labs
Palo Alto, US
benny@pinkas.net

Junichi Tomida
NTT Corporation
Tokyo, Japan
tomida.junichi@gmail.com

ABSTRACT

We implement a secure platform for statistical analysis over multiple organizations and multiple datasets. We provide a suite of protocols for different variants of JOIN and GROUP-BY operations. JOIN allows combining data from multiple datasets based on a common column. GROUP-BY allows aggregating rows that have the same values in a column or a set of columns, and then apply some aggregation summary on the rows (such as sum, count, median, etc.). Both operations are fundamental tools for relational databases. One example use case of our platform is in data marketing in which an analyst would join purchase histories and membership information, and then obtain statistics, such as "Which products were bought by people earning this much per annum?"

Both JOIN and GROUP-BY involve many variants, and we design protocols for several common procedures. In particular, we propose a novel group-by-median protocol that has not been known so far. Our protocols rely on sorting protocols, and work in the honest majority setting and against malicious adversaries. To the best of our knowledge, this is the first implementation of JOIN and GROUP-BY protocols secure against a malicious adversary.

KEYWORDS

Privacy-preserving protocols; multiparty computation; join; group-by; honest majority

1 INTRODUCTION

We construct a framework for collecting data from multiple sources, sharing it between multiple servers, and applying statistical analysis to the shared data. For this purpose, we design and implement various data analysis protocols that are actively secure in an honest majority setting. The inputs and outputs of these protocols are shared between the servers, and therefore different protocols can be easily composed for performing versatile data analysis.

We study two major types of protocols:

(1) Join protocols. Join allows to combine data from multiple datasets based on a common column (usually a key). This is a powerful tool for querying and analyzing relational databases. For example, in a financial system, Join can be used to combine data from a Transactions table, and an Accounts table. This can allow the system to display a customer's transaction history with details such as transaction amount, account balance, and transaction date. We design and implement various versions of join protocols, such as inner-join, outer-join, etc., and support the more challenging case of datasets that might have multiple entries with the same key.

(2) Group-by protocols group rows that have the same values in a certain column and then apply some aggregation or summary function on the rows of each group. For example, group-by-count computes the number of rows in each group. Group-by is particularly useful when dealing with large amounts of data because it allows condensing the data and generates summary statistics that provide useful insights. This is often used in data analytics applications, e.g., to identify trends or patterns. We design and implement protocols for computing group-by count/sum/max/min/median/percentile, where each of these protocols computes the designated statistic for each of the groups while hiding all other information about the groups.

1.1 Join Protocols

Considerable effort has been put into privately computing set intersection (PSI) and its variants. While most of these works concentrate on cases where the protocol outputs the intersection of the input sets, many applications require more general and robust properties:

- First, in addition to computing the intersection of two sets, it is necessary to compute more complicated functionalities, such as different types of join operations commonly used in SQL commands (right-join, left-join, etc.).
- It is useful to support a setting where both inputs and outputs of the protocol are secret-shared between multiple servers. This

enables the composition of different protocols and the computation of further joins, filtering, or aggregate information, without disclosing any intermediate results.
- Furthermore, many existing works assume that inputs (join keys) are unique, but this is not always the case with real-world data. Designing efficient protocols for non-unique keys is significantly harder, and techniques from the traditional setting of PSI cannot be automatically translated.

Join variants and settings. We briefly overview the various variants of join that we support. Let L and R be tables with n and m rows respectively. The first column in each table contains keys. The join operation outputs a new table in which each row is a concatenation of a row from L and a row from R. There are four common variants for the join operation. In all cases, the number of columns in the output table is the sum of the numbers of columns in L and in R.
- *Inner join*: The output table contains only rows with keys that appear in both tables. All rows in L and R that have the same key are concatenated and appear with the joint key.
- *Left outer join*: The output contains the inner join output table, and all rows from L which are unmatched (their key does not appear in R). These latter rows have null/0 in columns corresponding to R.
- *Right outer join*: The output contains the inner join output table, as well as all rows from R which are unmatched. These latter rows have null/0 in columns corresponding to L.
- *Full outer join*: The output contains the inner join table, plus all rows from both L and R that are not in the inner join. These latter rows have null/0 in columns corresponding to the other table.

A table can have a unique key for every row, or might have multiple rows with the same key. The join operation can have three different settings with regard to the keys:
- *uu:* Both tables have only *unique* keys (no duplicate keys).
- *un:* One of two tables can have duplicate keys. This case is typical when joining a table with attributes and a table with historical information. For example, each row in the first table includes a name and information about a person, such as an address and an age. In the second table, each row includes a name and the details of a purchase that this person has made, where a single person might have multiple rows associated with him or her.
- *nn:* Both tables can have duplicate keys. This is the general setting, which is also called many-to-many join, but is inherently less efficient. Our techniques do not directly apply to this setting, and we leave extending them to this setting as future work.

In Figure 1, we illustrate an example of an inner join-un protocol, i.e. an inner join where one table has unique keys (no duplicates) and the other table might contain multiple data rows with the same key. The output is the Table J. Note that the output table also contains rows with zeros elements. Such rows can be removed, as we will show later. However, this would reveal some information to the parties, namely, the size of the intersection. Thus, we view cleaning the zero rows as an optional computation on top of the output table. We will therefore provide an alternative definition for the different variants of join, that explicitly includes the null rows.

JOIN is a fundamental feature of the SQL language that is widely used in relational databases. It is also a standard SQL feature that

Input Table L
Attribute (no duplicate keys)

No.	Country	Age
3	USA	42
5	France	8
9	Canada	23

Input Table R
History (duplicate keys)

No.	Purchase
3	Fresh Water
7	Lemonade
9	Drink mix
9	Fresh Water

Table J output with the inner joined table

No.	Country	Age	Purchase
3	USA	42	Fresh Water
7	0	0	0
9	Canada	23	Drink mix
9	Canada	23	Fresh Water

Figure 1: Example of inner join-un (unique + non-unique). The inputs are tables L and R, the output is table J.

Input order

Key	Value
1	2
3	4
1	3
3	5
2	1

Grouped order

Key	Value
1	2
1	3
2	1
3	4
3	5

Output order

Key	Value
1	5
2	1
3	9
null	0
null	0

Figure 2: The three different orders when processing group-by-sum. The grouped order is a sorting of the input order according to the key, and the output order is after applying the aggregate function (sum) on each group, padded by null rows. (To improve readability, we do not use the notation $[[x]]$ to indicate that the values are shared between the parties.)

is supported by most relational database management systems, including popular systems like MySQL, Oracle, Microsoft SQL Server, PostgreSQL, and SQLite. JOIN is a ubiquitous feature in data analysis and reporting, both in relational databases and other data-related applications. In fact, JOIN is so essential to relational databases that it is difficult to imagine working with databases without it.

1.2 Group-By Protocols

We also study *group-by* operations over a shared table. Given a table containing elements of the form $\langle key, value \rangle$, where the same key might appear more than once, we group rows that have the same key into a summary record. This allows queries like "find the number of customers in each country". The "summary" part is an aggregate function such as COUNT, MIN, MAX, etc.

For ease of exposition, we have three different types of orders:
- **Input order:** This is the order of the rows received in the input. It contains keys and values in arbitrary order.
- **Grouped order:** This is sorting of the rows according to their keys. As such, all rows that have the same keys are located next to one another, and it is easier to compute aggregate functions on each group, such as COUNT, SUM, etc.
- **Output order:** After we apply the group-by operation, we obtain the output order. In that case, each key appears only once, and the "value" next to the key corresponds to the aggregated information. We fill the table to have the same dimensions as the input table with null values, to avoid leakage of the number of unique keys.

We give a concrete example for the GROUPBY.SUM in Figure 2.

We provide a suite of protocols for common Group-by operations, including COUNT, SUM, MIN, MAX, MEDIAN, PERCENTILE. Just like

Join, the main underlying technique is sorting, and the protocols were designed to utilize the linearity of the secret sharing scheme.

Just like JOIN, GROUP-BY is also a fundamental feature of the SQL language and is widely used in relational databases. It is an SQL feature that is supported by most relational database management systems. Furthermore, many data analysis and visualization tools, such as Excel, Tableau, and Power BI, also support Group-by clauses or similar functionalities. This makes Group-by a ubiquitous feature in data analysis and reporting, both in relational databases and other data-related applications.

1.3 Tools and Techniques

The main tool that we use is a secure protocol for sorting shared data, as in [4, 5]. We also use generic secure computation for computing operations such as multiplication and equality. Our protocols utilize the fact that linear operations on secret-shared data can be computed without any interaction, and minimize the usage of non-linear operations.

Overview of our techniques. We briefly overview some ideas behind our constructions.

Join. Assume that the left input table L contains only unique keys, and that the right input table R contains duplicate keys. Denote by $\mathbf{k}L$ (resp. $\mathbf{k}R$) the column of keys in L (resp. R). Moreover, the join variant that we are presenting in this overview is right-outer join. The join table should have the same number of rows as the R table, where for each key $\mathbf{k}R_i$ that appears in L, we append the corresponding row from L; if $\mathbf{k}R_i \notin L$ we append a row of zeros.

The difficulty is therefore in (1) locating keys in L that appear in R; and (2) (obliviously) propagating the corresponding values once found. Note that there is no upper bound on the number of times the same key might appear in R.

The protocol computes a permutation σ that (stable) sorts the list $(\mathbf{k}L, \mathbf{k}R, \mathbf{k}L)$. At this point, for every column L_i of L, we generate a vector $(L_i, 0^n, -L_i)$, where n is the number of rows in R. We apply σ to this vector. Suppose that a key k appears in $\mathbf{k}L$, and its row in L has the value v in column L_i. Denote by $|R(k)|$ the number of times this key appears in R. The permuted vector contains the sequence $(v, 0^{|R(k)|}, -v)$.

The protocol now computes a prefix-sum over the permuted vector. This is a linear operation and can therefore be computed without any communication. As a result of the prefix-sum, all the 0 values are replaced by the value v. The $-v$ at the end of the sequence cancels the value v, and propagates a 0 to the next position in the vector.

The protocol then applies the inverse permutation σ^{-1}. Each location in R that has a matching key in L receives the corresponding v value. All other locations in R receive a 0 value.

Group-by: A group-by protocol computes an aggregation function over all groups of rows, where a group is defined as a set of rows that have the same key. We implement group-by protocols for various aggregation functions, each requiring a designated protocol. In this overview, we focus on group-by-median, which was not studied before. The difficulty in computing group-by aggregation is that the values of the keys are shared and hidden from the servers.

Given a table T consisting of keys $\vec{k}$ and values $\vec{v}$, we sort the table according to the keys and move items to a grouped order, where rows with the same key are adjacent to each other. We present two novel protocols for computing ranks in an ascending order and in a descending order within each group. That is, if the group of rows with a key k has values $(10, 20, 30, 40, 50)$, then the ascending ranks are $\vec{a} = (0, 1, 2, 3, 4)$ and the descending ranks are $\vec{d} = (4, 3, 2, 1, 0)$. Note that 30 is the median for this group.

We then run a local computation to compute a new column to the table, which is $\vec{a} - \vec{d} = (-4, -2, 0, 2, 4)$. Note that only the median value, 30, has its corresponding value in $\vec{a} - \vec{d}$ equal to 0. By sorting once again, this time with a preference function that places 0s first and all other values afterward – we extract exactly the element 30.

The procedure we described only works if all groups have an odd number of elements. The full protocol computes the median of all groups, regardless of whether their size is even or odd.

Experiments. We implemented our join and group-by protocols in the three-party setting, using the recent efficient stable sorting protocol of Asharov et al. [5]. We ran experiments for various table sizes. Our results show that our protocols are highly efficient. For example, we are able to perform join-un for tables with 2^{20} rows in 5.4 seconds and 18 seconds for semi-honest and malicious security, respectively. For group-by median, which is the most time-consuming group-by operation, the output is computed in 4 seconds for semi-honest security and 12.5 for malicious security for a table of the same size (2^{20} records). Refer to Section 5 for more details.

1.4 Related Work

Blanton and Aguiar [9] proposed composable protocols for multiple set operations over secret-shared data in the honest majority setting. Their protocols are based on combining all inputs, sorting the combined list, and applying generic MPC to compute set operations. This structure for the protocol was used in different subsequent work, as well as in our work, while extending the supported operations and improving efficiency (which was $O(\log n)$ operations and communication in the original protocol). Laur et. al. [21] proposed a join protocol using oblivious shuffling and pseudo-random function (PRF). Their approach is to compute a (deterministic) oblivious PRF with the key as input, reveal it, and join tables of the same PRF output. The main drawback of this approach is that it leaks the number of duplicate keys and the sizes of intersections.

Mohassel et. al. [23] presented a practical join protocol based on cuckoo hashing and an oblivious switching network. On the downside, securing the last step against malicious adversaries appears to be challenging since it relies on one party not knowing the revealed value; furthermore, this protocol can only manage tables with *unique* keys. We also note that this protocol runs many secure computations of a random encoding. In order to improve performance, the implementation in [23] uses for this purpose the LowMC cipher [2], that was later cryptanalyzed [22]. Our protocols do not require MPC evaluation of specific ciphers, and therefore are not required to use extremely efficient ciphers that might have low security.

Sorting vs. hashing: It is instructive to compare the usage of sorting to another technique that was often used by protocols for private set intersection (PSI) (see e.g. [15, 25] and references within) – hashing items using cuckoo hashing or other hashing techniques so that all items that might potentially match with each

other are mapped to the same bin. There are two issues that make it difficult to use hashing in our setting: (1) PSI is typically computed between two parties that have private inputs, and therefore each party can independently compute the hashing over its own input items. On the other hand, in our setting, the inputs are shared, and thus hashing would have to be jointly computed by all servers using secure computation, which is much less efficient. (2) When all items have unique keys, applying a random hash function to the keys results in an output distribution that is independent of the inputs. On the other hand, in our setting, multiple items in an input set might have identical keys. In this case the output distribution of hashing might leak information about the distribution of the keys (at the extreme, if all input items have the same key then all items are hashed to the same bin).

GraphSC. The techniques employed in our join protocols bear a resemblance to those found in the GraphSC protocol and its subsequent developments [4, 24]. Specifically, as in those protocols our approach uses oblivious sorts for rearranging related elements to become adjacent. Afterwards information is disseminated between neighboring vertices. While GraphSC idea is conceptually similar, the setting of our work and that of GraphSC is significantly different: GraphSC is a framework for 2-party computation which is based on garbled circuits, whereas our work is for any number of parties and is based on secret sharing. Also, GraphSC achieves semi-honest security while we achieve malicious security.

Despite the clear distinction between the two works, we consider here a variant of GraphSC to the three-party case; Specifically, we consider a protocol in which oblivious sort is implemented in a similar manner as we do in our protocol. In addition, for computing a label indicating whether a value should be copied to the next stage, i.e., whether information should be propagated further, we use the prefix-sum as implemented in "parallel aggregate" (see [24, Fig 4]) while we consider this procedure as an interactive protocol and not as a Boolean circuit. Even if we adopt those ideas to the three parties setting, there exist two notable distinctions between our join protocol and the one that can be implied by GraphSC: Firstly, in our methodology, we apply an oblivious sort to a larger input size of $(2|L| + |R|)$ elements, as opposed to GraphSC's employment of oblivious sort on an input of size $|L| + |R|$. However, the benefit of our approach becomes apparent in the second difference: GraphSC requires additional $2\log(|L| + |R|)$ rounds and $O((|L| + |R|)\log(|L| + |R|))$ multiplications to compute the label indicating whether a value should be copied to the next stage. In contrast, such computation is not needed by our approach. This is because the information intended for propagation is inherently nullified due to the nature of our data organization. Specifically, each element in the set L appears twice–once with a positive value and once with a negative value—while the elements in the set R are positioned between these two instances. Consequently, we can effectively compute local prefix sum means without the concern of key matching. Using [5], the extra L elements in the oblivious sort is translated to about $O(|L|\ell_k) \approx O(|L|\log|L|)$ multiplications, where ℓ_k is the size of the keys. GraphSC needs an additional $O((|L| + |R|)\log(|L| + |R|))$ multiplications. If sorting is further improved, the additional multiplications in our protocol will decrease, whereas the additional multiplications in the GraphSC approach will remain the same.

Aggregation trees. The recent work of Badrinarayanan et. al. [8] presented efficient join protocols based on using secure sorting and *aggregation trees*. Aggregation trees apply to each element in a vector a function that depends on the previous elements in the vector. An aggregation tree is computed by computing two passes up and down a tree that is constructed over the data. (This means that computing an aggregation tree requires about $2\log n$ rounds, besides the calls to the oblivious sorts.)

The aggregation tree construction has origins in the Parallel RAM (PRAM) literature, where it was denoted as "prefix sum" or "segmented scan" [10, 11]. We use the fact that secure computation of linear operations, such as summation, can be done locally. Therefore, unlike the PRAM literature, we are not bothered by the number of "rounds" required for computing linear functions. We were thus able to replace the usage of aggregation trees by applying secure sorting together with simple secure computation and local computation of linear functions. The usage of sorting is preferable over aggregation trees since it requires about $2\log(|L|+|R|)$ less communication rounds, and, perhaps more importantly, does not require implementing a new algorithm, since sorting is already used by the protocols. In addition, it seems non-trivial to utilize aggregation trees for computing the group-by-median or group-by-percentile functionalities.

For comparison, our protocol requires running oblivious sort on a larger input, while the aggregation tree approach [8] requires extra $2\log(|L|+|R|)$ rounds and extra $O(|L|+|R|)$ multiplications beyond those that are necessary for the oblivious sort. The protocol in [8] describes only inner joins.[1] Furthermore, no secure protocol against malicious adversaries is proposed (the paper says that the protocol can be extended to the malicious setting, but due to subtleties about making generic statements in the malicious setting (input extraction), this is deferred to future work). On the other hand, the work of [8] also supports non-unique keys, while ours does not.

We summarize the differences between our work and GraphSC [24] and aggregation trees [8] in Table 1. We note that protocols based on Radix sort (such as [5]) require communication rounds linear in a bit of the key. Therefore, even though we sort $2|L| + |R|$ elements, the round complexity is the same as sorting $|L| + |R|$ elements — since the length of the keys is the same in both cases. Therefore, our round complexity corresponds to the length of the keys, whereas the round complexity of the other protocols is larger by at least $\log(|L| + |R|)$ rounds.

Group-by-sum operation. Hamada et al. [17] proposed a protocol for computing decision trees, and along the way computed group-by-sum, while the input/output format is different, tailored for decision trees.

Although we can construct a group-by-sum protocol whose input and output format are the same as ours by modifying their protocol accordingly, there is a difference in efficiency. Our work focuses on computing multiple statistics simultaneously using group-by operations, and our protocols are optimized for these cases by

[1] The paper [8] claims that other joins can be computed in the same way as in [23], but it is unclear how to apply this for left/full outer join because in these cases, the output must contain rows that were removed earlier in the protocol.

Protocol	Oblivious Sort on input	Extra Rounds	Extra Multiplications
[24]	$n+m$	$\log(n+m)$	$O((n+m)\log(n+m))$
[8]	$n+m$	$2\log(n+m)$	$O(n+m)$
Ours	$2n+m$	0	0

Table 1: Comparison of our work joint protocol with [8], and an adpation of GraphSC [24] for the three party setting, where $|L| = n$ and $|R| = m$. All protocols are based on oblivious sort, while ours needed no extra rounds or multiplications beyond the oblivious sort.

reusing a common operation (called a group-by-common operation, described in Section 4.1). In case of computing multiple statistics at once, our protocol is more efficient and requires one less call to a sorting function compared to [17]. If the only statistics required are just group-by-sum, then the two protocols have similar costs. A more detailed comparison is discussed in Section 4.3.

Other works. There have been multiple works on grouped statistics on secret shares [3, 4, 7, 17, 24]. The most relevant result of Attrapadung et. al. [7], describing a grouped aggregation protocol that can support group operations which include sum, max, and min. The main difference is that our work also supports the median and percentile group operations. In addition, as in computing the join operation, our protocol only requires black-box computations of sorting and simple circuits, making it easier to implement. In fact, no implementation or evaluation is reported in [7].

Searchable encryption [14, 27] is a technique for outsourcing information to a remote server and later extracting information based on keyword search. Searchable encryption is based on the model in which there is only one server. The common techniques leak information, such as the search pattern or the access pattern. In contrast, our model requires three servers, which allows for hiding such information and efficiently computing more complex functions (like group-by operations) on the servers' side.

Summary of our contributions. We conclude this section by providing a summary of our contributions:

- We provide a suite of protocols for statistical analysis on multiple databases, supporting two major queries: JOIN and GROUP-BY.
- We support the main variants of JOIN: inner join, left outer join, right outer join, and full outer join. We also support the challenging case where one of the tables has non-unique keys.
- We support computing the GROUP-BY operation for aggregate computation over groups of COUNT, SUM, MIN, MAX, MEDIAN, and PERCENTILE. To the best of our knowledge, ours is the first technique that allows processing GROUP-BY of MEDIAN and PERCENTILE.
- All our protocols are secure against malicious behavior, and work in the honest majority setting.
- Implementation: We have implemented our protocols in the three-party setting. To the best of our knowledge, this is the first implementation of join and group-by protocols secure against a malicious adversary.

2 PRELIMINARIES

Let N denote the number of parties running the MPC protocol, and t be the corruption threshold. In this work, we assume an honest majority exists, i.e., $t < \frac{N}{2}$. Let $\vec{a} = (a_1, \ldots, a_m)$ and $\vec{b} = (b_1, \ldots, b_m)$ be vectors. Let $||$ denote the concatenation of vectors, i.e., $\vec{a}||\vec{b} = (a_1, \ldots, a_m, b_1, \ldots, b_m)$. We use $\mathbb{F}_p$ to denote a finite field, where all operations are modulo a prime number p.

Security definition. To prove that our protocols are secure, we use the standard definition based on the ideal/real-world paradigm [16]. We consider malicious security formalized for non-unanimous abort. This means that the adversary first receives the output and then determines for each honest party whether they will receive abort or receive their correct output.

Linear secret sharing schemes. A secret sharing scheme with threshold t allows to distribute a secret across multiple parties with the following two properties: (i) Each subset of t parties cannot learn any information about the secret; (ii) Any subset of $t+1$ honest parties can reconstruct the secret from the shares that they received. An additional property that we require from the scheme is *linearity*, namely that given threshold t sharings of two secrets a and b, if each user computes the sum of the shares it received for a and b, we obtain a threshold t secret sharing of $a+b$. Linear secret sharing implies that linear operations, such as addition or multiplication-by-a-constant, can be done locally by the parties over their shares of the secret without any interaction. Two schemes that satisfy these properties and are commonly used in the honest majority setting are the Shamir's secret sharing scheme [26] and the replicated secret sharing scheme [18].

In this paper, we use the following two notations:

- $[\![a]\!]$: a secret sharing of $a \in \mathbb{F}_p$.
- $[\![\vec{a}]\!]$: a secret sharing of a vector $\vec{a}$, i.e., $[\![\vec{a}]\!] = ([\![a_1]\!], \ldots, [\![a_m]\!])$.

Basic procedures. We next define some basic procedures that are used in our protocols:

- $[\![\vec{a'}]\!] \leftarrow$ PrefixSum$([\![\vec{a}]\!])$: Given a secret sharing $\vec{a} = a_1, \ldots a_m$ as an input, output a secret sharing of $\vec{a}' = a'_1, \ldots, a'_m$, where $a'_k = \sum_{i=1}^{k} a_i$.
  By the linearity of the secret sharing scheme, this procedure can be locally computed by the parties with no interaction.
- $[\![\beta]\!] \leftarrow$ Equal$([\![a]\!], [\![b]\!])$: If $a = b$, output a secret sharing of $\beta = 1$. Otherwise, output a secret sharing of $\beta = 0$. Similarly, we define $[\![\beta]\!] \leftarrow$ Equal$([\![a]\!], c)$ that outputs $\beta = 1$ if $c = a$ and 0 otherwise These procedures can be implemented through bit decomposition [20] and a circuit computation for checking bit equality.
- $[\![\beta]\!] \leftarrow$ LessThan$([\![a]\!], [\![b]\!])$: If $a < b$, output a secret sharing of $\beta = 1$. Otherwise, output a secret sharing of $\beta = 0$.
  This procedure can be implemented by first applying bit decomposition and then computing a circuit which compares the bit representation of each element.
- $[\![out]\!] \leftarrow$ Ifthen$([\![in]\!] : t, f)$): Given $in \in \{0, 1\}$, if $in = 1$, $out = t$; otherwise, $out = f$. Similarly, we define $[\![out]\!] \leftarrow$ Ifthen$([\![in]\!] : [\![t]\!], [\![f]\!])$ that outputs the same whereas the inputs are shared.

These procedures can be implemented through circuit computation as $[[out]] = [[in]] \cdot ([[t]] - [[f]]) + [[f]]$ (i.e., one multiplication).

- $a \leftarrow$ Reconstruct$([[a]])$: In this procedure, the parties reveal a secret a by sending their shares of a to each other. In the honest majority setting, the shares of the honest parties alone suffice for reconstructing the secret. Hence, the adversary cannot change the opened value but only cause an abort.

Ideal functionalities. We define several ideal functionalities that are used in our protocols.

(1) $\mathcal{F}_{mult}$*- multiplication ideal functionality:* We define an ideal functionality $\mathcal{F}_{\text{mult}}$ for multiplying shared values. The functionality receives $[[x]]$ and $[[y]]$ from the honest parties, reconstruct x and y, and share $z = x \cdot y$ to the parties. If the adversary is malicious, then $\mathcal{F}_{\text{mult}}$ first hands it its shares on $[[x]]$ and $[[y]]$, and it allows the adversary to choose the corrupted parties' shares of z.

(2) $\mathcal{F}_{zero}$, $\mathcal{F}_{one}$ *- Random secret generation:* Let $\mathcal{F}_{\text{zero}}$ be an ideal functionality that hands the parties a random secret sharing of 0, while letting the adversary choose the corrupted parties' shares. For a small number of parties, this functionality can be realized efficiently without any interaction (except for short setup) via *pseudorandom zero sharing*; e.g.,[13]. Observe that the parties can generate a secret sharing of any constant c by generating $[[0]]$ and then locally adding c. In the paper, we make use of the ideal functionality $\mathcal{F}_{\text{one}}$, which produces a secret sharing of 1.

(3) $\mathcal{F}_{sort}$ *- The sorting ideal functionality:* A permutation σ is a bijective function from a finite set to itself, where usually the underlying set is $[m] = \{1, \ldots, m\}$. A main building block in our protocol is secure sorting, i.e., applying a permutation that sorts a set of items, without revealing any information about the set or the sorting permutation. To this end, let $\vec{v} = (v_1, \ldots, v_m)$ be a vector of items. Applying a permutation σ over $\vec{v}$ means that the item v_i is moved to position $\sigma(i)$. We abuse notation and use $\sigma(\vec{v})$ to denote applying σ on $\vec{v}$, which results with a vector $\vec{u}$. We use $\vec{\sigma} = (\sigma(1), \ldots, \sigma(m))$ to denote the vector of destinations of σ.

The $\mathcal{F}_{\text{sort}}$ ideal functionality defined below has three commands: GenPerm which computes the permutation that sorts a given input vector $[[\vec{k}]]$; `ApplyPerm` which applies the permutation on a shared vector, and `ApplyInv` which applies the inverse of the permutation on a shared vector. See Functionality 2.1. An implementation for this functionality for the three-party setting is described in [5]. The number of communication rounds in `GenPerm` is the same as the bit-length of the keys, which we can assume to be $O(\log n)$. `ApplyPerm` and `ApplyInv` have a constant number of communication rounds. All our JOIN protocols compute `GenPerm` at most once, and therefore the number of communication rounds is the length of the key plus $O(1)$.

Functionality 2.1 ($\mathcal{F}_{\text{sort}}$ – Stable sorting)**:**

- **GenPerm:** Upon receiving (`GenPerm`, $[[\vec{k}]]$) from the honest parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{k}$, and computes the permutation σ such that $\sigma(i) \leq \sigma(j)$ if $k_i \leq k_j$ and $i < j$. Then, it generates $[[\vec{\sigma}]]$ and sends each party its shares.
- **ApplyPerm:** Upon receiving (`ApplyPerm`, $[[\vec{\sigma}]]$, $[[\vec{v}]]$) from the honest parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{v}$ and the permutation σ and applies $\sigma(\vec{v})$ to obtain $\vec{v}'$. It computes shares of $\vec{v}'$ and sends each party its shares.
- **ApplyInv:** Upon receiving (`ApplyInv`, $[[\vec{\sigma}]]$, $[[\vec{v}]]$) from the honest parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{v}$ and the permutation σ and applies the inverse of $\sigma(\vec{v})$ to obtain $\vec{v}'$. It computes shares of $\vec{v}'$ and sends each party its shares.

If the adversary is malicious, then $\mathcal{F}_{\text{sort}}$ allows it to choose the corrupted parties' shares in each of the above.

3 SECURE JOIN PROTOCOLS

We present efficient protocols for securely joining two tables that are shared across a set of parties. The different variants and settings for the join task were described in the introduction (Section 1.1). We present protocols for each of these variants.

The join protocols use a sub-protocol denoted FromLtoR. The functionality of this protocol is defined in Section 3.2, and its implementation is described in Section 3.5.

3.1 Join-un

We first discuss joining two tables, where one table contains unique keys and the other table may contain duplicate keys. The protocol makes direct usage of sorting and simple arithmetic operations and does not require implementing an aggregation tree.

Figure 3 illustrates the input and output of the join protocol.

Input Table L

Key	Values			Valid
$[[\mathbf{k}L_1]]$	$[[L_{1,1}]]$	$\cdots$	$[[L_{1,d}]]$	$[[\mathbf{v}L_1]]$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$[[\mathbf{k}L_m]]$	$[[L_{m,1}]]$	$\cdots$	$[[L_{m,d}]]$	$[[\mathbf{v}L_m]]$

Input Table R

Key	Values			Valid
$[[\mathbf{k}R_1]]$	$[[R_{1,1}]]$	$\cdots$	$[[R_{1,e}]]$	$[[\mathbf{v}R_1]]$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$[[\mathbf{k}R_n]]$	$[[R_{n,1}]]$	$\cdots$	$[[R_{n,e}]]$	$[[\mathbf{v}R_n]]$

Output Table J (the result of the join of Table L and R)

Key	Values from L			Values from R			Valid
$[[\mathbf{k}J_1]]$	$[[JL_{1,1}]]$	$\cdots$	$[[JL_{1,d}]]$	$[[JR_{1,1}]]$	$\cdots$	$[[JR_{1,e}]]$	$[[\mathbf{v}J_1]]$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$[[\mathbf{k}J_{n'}]]$	$[[JL_{n',1}]]$	$\cdots$	$[[JL_{n',d}]]$	$[[JR_{n',1}]]$	$\cdots$	$[[JR_{n',e}]]$	$[[\mathbf{v}J_{n'}]]$

Figure 3: Inputs and output of join algorithm.

The two tables to be joined are denoted as L and R, with Table L having m rows and $d + 2$ columns and Table R having n rows and $e + 2$ columns:

- **Keys:** The first column in both Table L and Table R contains the keys, denoted as $[[\vec{\mathbf{k}L}]] = ([[\mathbf{k}L_1]], \ldots, [[\mathbf{k}L_m]])$ for the left table L and $[[\vec{\mathbf{k}R}]] = ([[\mathbf{k}R_1]], \ldots, [[\mathbf{k}R_n]])$ for R. We assume that $\mathbf{k}L_i, \mathbf{k}R_j$ are positive integers for all i, j. It is assumed that the keys of the left table, $[[\vec{\mathbf{k}L}]]$, are all unique, whereas the keys of the right table $[[\vec{\mathbf{k}R}]]$ may have duplicate values.
- **Values:** In addition to the keys, the tables contain other entries, where the entry of row i and column j is denoted as $[[L_{i,j}]]$ and $[[R_{i,j}]]$, in tables L and R, respectively.
- **Validity:** The last columns of both tables consist of bits that represent whether each row is valid (the bit is 1) or not (the bit is 0). The validity bit of the ith row in table L or R, is denoted as $\mathbf{v}L_i$ or $\mathbf{v}R_i$, respectively.

If the bit is invalid, it means that the row is not part of the table, and any values in the row should be ignored.

The output of the join operation is Table J, and depends on what kind of join is performed. The format of J is basically the same as of the input tables; the first column consists of keys, the last column consists of validity bits, and the rest of the columns consist of other entries. A caveat is that in the output of our protocols, the values $(JL_{i,1}, \ldots, JL_{i,d})$ and $(JR_{i,1}, \ldots, JR_{i,e})$ may not be 0 (or null) even if $\mathbf{v}J_i = 0$. Hence, in order to use the output table as the input table of our join-un protocols, we need to correct the format of the output table such that $(JL_{i,1}, \ldots, JL_{i,d})$ and $(JR_{i,1}, \ldots, JR_{i,e})$ are all 0 if $\mathbf{v}J_i = 0$. This can be easily obtained by multiplying the row with $\mathbf{v}J_i$.

Recall that we are computing a join-un and that table L has m rows and table R has n rows. In order to hide the number of rows in the output, the number n' of rows in the joined table is set to be n for an inner join and a right outer join (since in these cases the output table might have as many rows as R). For a left outer join and a full join, the number n' of rows is set to be $m + n$.

In Section 3.11, we show how to remove "null rows"; however, for now we present the tables as containing null rows as this also makes the protocols simpler. We now define the tables that are result by our operations:

Definition 3.1 (The table J). *Let L be a $m \times (d+2)$ table, and R be $n \times (e+2)$ table, as described in Figure 3. Then, a join operation between L and R outputs a table J, where the i-th row of J consists of*

$$([[\mathbf{k}J_i]], [[JL_{i,1}]], \ldots, [[JL_{i,d}]], [[JR_{i,1}]], \ldots, [[JR_{i,e}]], [[\mathbf{v}J_i]])$$

and is defined as described in Table 2 (for an inner join).

3.2 From L to R

A sub-procedure, which is also the crux of our protocols, is the protocol FromLtoR. The procedure receives as input sharing of the keys $[[\vec{\mathbf{k}L}]]$, $[[\vec{\mathbf{k}R}]]$, and a sharing of a vector $[[\vec{A}]] = ([[A_1]], \ldots, [[A_m]])$, which we can think of as corresponding to some column of L. The procedure outputs a sharing of a vector $([[x_1]], \ldots, [[x_n]]) = \mathsf{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{A}]])$, such that :

$$x_i = \begin{cases} A_j & \text{If } \exists j \text{ s.t. } \mathbf{k}L_j = \mathbf{k}R_i \\ 0 & \text{otherwise} \end{cases}.$$

In other words, the output is a sharing of a vector with the same size as a column of R; Moreover, it contains a value from $[[\vec{A}]]$ if the corresponding keys in L and R match. (Recall that the L has unique keys, and therefore there is at most a single match in L for any key in R.) If we invoke this procedure where $\vec{A}$ is the ith column of L, we get precisely the column JL_i in J in, e.g., an inner join. As another example, if we invoke the procedure on $[[\vec{A}]]$ which is the all one vector, we receive as an output a column that consists of indicators whether the "condition" column in Table 2 holds or not.

Looking ahead, when performing, e.g., a right outer join, the parties invoke this procedure the same number of times as the number of columns in L. We still do not show how to implement FromLtoR but only describe next the two variants of right outer join and inner join using FromLtoR as a sub-procedure. In Section 3.5, we show how to implement the FromLtoR protocol.

3.3 Right Outer Join

Intuitively, to compute the right outer join, the parties invoke FromLtoR on each of the columns of the table L. This results in the columns $JL_1, \ldots, JL_d$ in J of right outer join as per Table 2. The parties output the column of the key of R, $\mathbf{k}R$, followed by the computed columns $JL_1, \ldots, JL_d$, followed by the columns $JR_1, \ldots, JR_e$ and the validation column $\mathbf{v}R$. Given the correctness of the FromLtoR procedure, the correctness of the right outer join is immediate. We describe the protocol in Protocol 3.2.

Protocol 3.2 ($[[J]] \leftarrow$ RightOuterJoin($[[L]]$, $[[R]]$)):

Input: (Shares of) tables $[[L]]$ and $[[R]]$ shown in Figure 3 (both tables are assumed to be sorted according to their keys).
Output: (Shares of) Table $[[J]]$ as defined in Definition 3.1 for the right outer join operation.
The protocol:
(1) For every $k \in [d]$, invoke

$$[[\vec{JL_k}]] = ([[JL_{k,1}]], \ldots, [[JL_{k,n}]]) = \text{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{L_k}]]),$$

where L_k denotes the kth column of L. (see Section 3.2).
(2) **Output:** $J = ([[\mathbf{k}R_i]], [[JL_{k,1}]], \ldots, [[JL_{k,n}]], [[R_i, 1]], \ldots, [[R_i, e]], [[\mathbf{v}R_i]])_{1 \le i \le n}$.

Note that the cost of this protocol is basically the cost of FromLtoR, which we analyze in Section 3.5.

3.4 Inner Join

For an inner join, the parties first compute the right outer join. Then, for each row in R for which there is no corresponding key in L, we have to replace the entire row with 0 in the table J. We do that by invoking FromLtoR on a vector that is all 1. This gives as output a vector $(c_1, \ldots, c_n)$, which is exactly the condition column in Table 8. That is, $c_i = 1$ iff $\exists j$ such that $\mathbf{k}L_j = \mathbf{k}R_i$. We then nullify the corresponding rows of R by using $\mathcal{F}_{\text{mult}}$ and multiplying the ith row of R with c_i. Finally, we compute the validation vector $\mathbf{v}J$ by invoking FromLtoR on the vector $\mathbf{v}L$, and then pairwise multiply (using $\mathcal{F}_{\text{mult}}$) the result with $\mathbf{v}R$. This results in $\mathbf{v}L_j \wedge \mathbf{v}R_i$ if there exists j s.t. $\mathbf{k}L_j = \mathbf{k}R_i$, or 0 otherwise. The protocol is described in Protocol 3.3.

3.5 From L to R – Implementation

We now turn our attention to the implementation of the procedure FromLtoR. The protocol is described in Protocol 3.4. The protocol receives as input the keys of the tables $[[\vec{\mathbf{k}L}]]$ and $[[\vec{\mathbf{k}R}]]$. It first computes a permutation that computes a stable sort of $[[\vec{\mathbf{k}L}]]$, $[[\vec{\mathbf{k}R}]]$, $[[\vec{\mathbf{k}L}]]$ (Note that the vector $[[\vec{\mathbf{k}L}]]$ appears twice, before and after $[[\vec{\mathbf{k}R}]]$.) It applies this permutation to a shared vector $([[A_1]], \ldots, [[A_m]])$, $[[0]]^n$, $([[-A_1]], \ldots, [[-A_m]])$, where the middle part of the vector, $[[0]]^n$, represents the entries of R, and the outer parts are the entries of L. As a result, a 0 corresponding to a row in R which has the same key as a row in L, is located between the A_i and $-A_i$ values corresponding to that row in L. The protocol now computes a prefix sum. As explained below, this computation replaces any 0 entry whose key matches a key in L with the corresponding A_i value. Finally, the protocol applies the reverse of the

Join Type	*Key* $\mathbf{k}J_i$	*Values from L* $JL_{i,1}$	…	$JL_{i,d}$	*Values from R* $JR_{i,1}$	…	$JR_{i,e}$	*Valid* $\mathbf{v}J_i$	**Condition**
Right outer join	$\mathbf{k}R_i$	$L_{j,1}$	…	$L_{j,d}$	$R_{i,1}$	…	$R_{i,e}$	$\mathbf{v}R_i$	$\exists j$ s.t. $\mathbf{k}L_j = \mathbf{k}R_i$
	$\mathbf{k}R_i$	0	…	0	$R_{i,1}$	…	$R_{i,e}$	$\mathbf{v}R_i$	otherwise
Inner join	$\mathbf{k}R_i$	$L_{j,1}$	…	$L_{j,d}$	$R_{i,1}$	…	$R_{i,e}$	$\mathbf{v}L_j \wedge \mathbf{v}R_i$	$\exists j$ s.t. $\mathbf{k}L_j = \mathbf{k}R_i$
	$\mathbf{k}R_i$	0	…	0	0	…	0	0	otherwise
Left outer join	*For $i \le n$, left outer join is the same as inner join; For $n < i < n+m$:*								
	$\mathbf{k}L_i$	$L_{i,1}$	…	$L_{i,d}$	0	…	0	$\mathbf{v}L_i$	$(\forall j : \mathbf{k}L_i \neq \mathbf{k}R_j)$
	$\mathbf{k}L_i$	0	…	0	0	…	0	0	*otherwise*
Full join	*For $i \le n$, full join is the same as inner join* *For $n < i \le n+m$, full join is the same as in left outer join*								

Table 2: Definition of the output table for the four join types, for unique (in L)- non-unique (in R) – un.

Protocol 3.3 ($[[J]] \leftarrow \textsc{InnerJoin}([[L]], [[R]])$):

Input: (Shares of) tables $[[L]]$ and $[[R]]$ shown in Fig. 3 (both tables are assumed to be sorted according to their keys).
Output: (Shares of) Table $[[J]]$ as defined in Definition 3.1 for the inner-join operation.
The protocol:

(1) For every $k \in [d]$, The parties invoke $[[J\vec{L}_k]] = ([[JL_{k,1}]], \ldots, [[JL_{k,n}]]) = \textsc{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{L_k}]])$, where L_k denotes the kth column of L (see Section 3.2).
(2) **Computing $JR_1, \ldots, JR_e$ columns**:
  (a) Generate a vector $[[1]]^m$ by invoking $\mathcal{F}_{\text{one}}$ m times.
  (b) Invoke $[[\vec{c}]] = \textsc{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[1]]^m)$. Note that $c_i = 1$ iff there exists j such that $\mathbf{k}L_j = \mathbf{k}R_j$.
  (c) For every $i \in [n]$ and $j \in [e]$: the parties send $([[c_i]], [[R_{i,j}]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[JR_{i,j}]]$.
(3) **Computing $\mathbf{v}J$ column:**
  (a) Invoke $[[\vec{v}]] = \textsc{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{\mathbf{v}L}]])$, where $v_i = 1$ if there exists j such that $\mathbf{k}L_j = \mathbf{k}R_i$ and $\mathbf{v}L_j = 1$.
  (b) for every $1 \le i \le d$, send $([[v_i]], [[\mathbf{v}R_i]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[\mathbf{v}J_i]]$.
(4) **Output:** $J = ([[\mathbf{k}R_i]], [[JL_{i,1}]], \ldots, [[JL_{i,d}]], [[JR_{i,1}]], \ldots, [[JR_{i,e}]], [[\mathbf{v}J_i]])_{1 \le i \le n}$.

sorting permutation to return the vector entries to their original locations.

An execution of Protocol 3.4. To see why this procedure works, we demonstrate the execution of the protocol inside the inner join protocol for the example given in Figure 1. Specifically:

The permutation $\vec{\sigma}$: The parties start with shares $[[\vec{\mathbf{k}L}]]$ and $[[\vec{\mathbf{k}R}]]$, and compute a permutation of a vector that contains the keys of L, followed by the keys of R, and then followed again by the keys of L. Note that the keys of Table L *appear twice* at the left and right sides of the unordered input. The keys of R appear between the two copies of the keys of L. Although we do not apply σ to the data, applying σ to the inputs would reorder keys as follows:

Input:	[3]	[5]	[9]	(3)	(7)	(9)	(9)	[3]	[5]	[9]
	L3	L5	L9	R3	R7	R9	R9	-L3	-L5	-L9
Output:	[3]	(3)	[3]	[5]	[5]	(7)	[9]	(9)	(9)	[9]
	L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

Protocol 3.4 ($[[\vec{x}]] \leftarrow \textsc{FromLtoR}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{A}]])$):

Input: (Shares of) the keys $[[\vec{\mathbf{k}L}]]$, $[[\vec{\mathbf{k}R}]]$, i.e., the keys of the left table and the right table, respectively. In addition, shares of a vector $A = (A_1, \ldots, A_m)$.
Output: (Shares of) a vector $[[\vec{x}]] = ([[x_1]], \ldots, [[x_n]])$, where $x_i = A_i$ if there exists $\mathbf{k}L_j = \mathbf{k}R_i$, and 0 otherwise.
The protocol:

(1) The parties send $(\texttt{GenPerm}, ([[\vec{\mathbf{k}L}]] \parallel [[\vec{\mathbf{k}R}]] \parallel [[\vec{\mathbf{k}L}]]))$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{\sigma}]]$.
(2) The parties generate n shares of $[[0]]$ (via calls to $\mathcal{F}_{\text{zero}}$). Recall that n is the number of rows in the table R.
(3) Consider the vector $[[\vec{f}]]$ defined as
$$([[A_1]], \ldots, [[A_m]]), [[0]]^n, ([[-A_1]], \ldots, [[-A_m]]) .$$
(4) The parties send $(\texttt{ApplyPerm}, [[\vec{\sigma}]], [[\vec{f}]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{g}]]$.
(5) $[[\vec{h}]] \leftarrow \textsc{PrefixSum}([[\vec{g}]])$ (see Section 2).
(6) The parties send $(\texttt{ApplyInv}, [[\vec{\sigma}]], [[\vec{h}]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{f'}]] = ([[f'_1]], \ldots, [[f'_{2m+n}]])$.
(7) **Output:** $[[f'_{m+1}]], \ldots, [[f'_{m+n}]]$.

Here, we use the notation $\boxed{\!\boxed{x}\!}$, $\textcircled{y}$, $\boxed{x}$, where $\boxed{\!\boxed{x}\!}$ and $\boxed{x}$ represent the two copies of the key of Table L, and $\textcircled{y}$ is the key of Table R. Under each element, we also present in gray LY or RY, where the key is Y and the table the element comes from either L or R. Elements in the second copy of L are differentiated using a minus sign (−). We use this notation to indicate the location relationships before and after the permutation and to differentiate identical values.

We demonstrate the execution of the protocol FromLtoR on the column "Age" in the example of Figure 1. In this process, the vector $\vec{f}$ defined in Step 3 is

$\vec{f} =$	$\boxed{\!\boxed{42}\!}$	$\boxed{\!\boxed{8}\!}$	$\boxed{\!\boxed{23}\!}$	(0)	(0)	(0)	(0)	$\boxed{-42}$	$\boxed{-8}$	$\boxed{-23}$
	L3	L5	L9	R3	R7	R9	R9	-L3	-L5	-L9

where in gray, we show the corresponding keys. Next, in Step 4, we compute the vector $\vec{g} = \texttt{ApplyPerm}(\sigma, \vec{f})$, followed by $\vec{h} = \text{PrefixSum}(\vec{g})$ in in Step 5:

$\vec{g}$ =

⟦42⟧	(0)	[-42]⟧	⟦8⟧	[-8]⟧	(0)	⟦23⟧	(0)	(0)	[-23]⟧
L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

$\vec{h}$ =

⟦42⟧	(42)	[0]⟧	⟦8⟧	[0]⟧	(0)	⟦23⟧	(23)	(23)	[0]⟧
L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

Then, in Step 6, we invoke the inverse permutation on the result, which returns each element to its original location, and we obtain $\vec{f}'$. In our example, $\vec{f}'$ is

$\vec{f}'$ =

⟦42⟧	⟦8⟧	⟦23⟧	(42)	(0)	(23)	(23)	[0]⟧	[0]⟧	[0]⟧
L3	L5	L9	R3	R7	R9	R9	-L3	-L5	-L9

We claim that the vector "in the middle", that corresponds to the vector of elements that originated in R, that is, the vector $(42, 0, 23, 23)$,is precisely the "Age" column in the output table J (see Figure 1).

More generally, if the input to the procedure is some vector of values $(A_1, \ldots, A_m)$, then denoting the vector $\vec{f}'$ as $(f'_1, \ldots, f'_{2m+n})$, we claim that the values "in the middle" $(f'_{m+1}, \ldots, f'_{m+n})$ satisfy:

$$f'_{m+i} = \begin{cases} A_j & \text{If } \exists j \text{ s.t. } \mathbf{k}L_j = \mathbf{k}R_i \\ 0 & \text{otherwise} \end{cases} .$$

To see why this holds, consider the vector $\vec{g}$ that is obtained after applying σ. We divide $\vec{g}$ into sub-sequences and claim that in $\vec{h}$ at the end of each sub-sequence, the prefix sum 0. As a result, we can analyze each sub-sequence separately. We have three types of sub-sequences:

(1) **Type I: sub-sequence of the form** $[\![A_j]\!], (0), \ldots, (0), [-A_j]\!]$: This sub-sequence occurs if there exists a key $\mathbf{k}L_j$ in L that appears in R. The number of (0)s then correspond to the number of times the key $\mathbf{k}L_j$ appears in R. Moreover, we want each such (0) to obtain the value A_j since this is the value associated with $\mathbf{k}L_j$ in the vector $\vec{A}$. Assuming that the prefix sum when reaching this sub-sequence starts is 0, the corresponding prefix-sum in $\vec{h}$ is $([\![A_j]\!], (A_j), \ldots, (A_j), [0]\!])$, i.e., the last element is 0. Moreover, each element from R with key $\mathbf{k}R_i$, for which there exists $\mathbf{k}L_j$ in L, receives the value A_j.

(2) **Type II: sub-sequence of the form** $[\![A_j]\!], [-A_j]\!]$: This sub-sequence occurs if the key $\mathbf{k}L_j$ does not appear in R. In $\vec{h}$, assuming that the prefix sum when reaching this sub-sequence is 0, then corresponding prefix sum in $\vec{h}$ is $([\![A_j]\!], [0]\!])$. The prefix sum of the last element in the sub-sequence is 0.

(3) **Type III: sub-sequence of the form** $(0), \ldots, (0)$**:** (which continues until viewing the element $[\![A_k]\!]$). This sub-sequence occurs when there are rows in R with key $\mathbf{k}R_i$ that do not exist in L. Assuming that the prefix sum in $\vec{h}$ until reaching this sub-sequence is 0, the sum remains 0 since all elements in this sub-sequence are 0s.

Besides the fact that the prefix of each sub-sequence ends with 0, we have that for each value originated in R:

(1) If there exists a key $\mathbf{k}L_j$ in L such that $\mathbf{k}L_j = \mathbf{k}R_i$ (i.e., Type I), then the corresponding value in R in $\vec{h}$ obtains the value A_j.

(2) If there does not exist a key $\mathbf{k}L_j$ in L such that $\mathbf{k}L_j = \mathbf{k}R_j$, then the value originated in R obtains the value 0 in $\vec{h}$ (Type III).

Thus, after "unsorting", the elements return to their original places in $\vec{f}$. As a result, each element f'_{m+i} (which is originally (0) in $\vec{f}$), is either A_j if $\mathbf{k}L_j = \mathbf{k}R_i$, or 0 otherwise.

Implementation note: In our join protocols, we invoke FromLtoR several times, where in all invocations, the input contains the same keys $\mathbf{k}L, \mathbf{k}R$. Therefore there is no need to re-compute the permutation $\vec{\sigma}$ every time, as the protocol can re-use the computed permutation from previous invocations.

3.6 Only in L

Before moving to the left and full join operations, we provide a sub-procedure that is needed in order to process left outer and full joins. Specifically, now we are interested in finding the keys in L that appear only in L and do not have corresponding keys in R. More formally, we are interested in a procedure.

$$[[\vec{c}]] = \text{OnlyInL}([[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]])$$

with $[[\vec{c}]] = (c_1, \ldots, c_m)$, such that:

$$c_i = \begin{cases} 1 & \text{if } (\forall j \ : \ \mathbf{k}L_i \neq \mathbf{k}R_j) \\ 0 & \text{otherwise} \end{cases} ,$$

that is, $c_i = 1$ if the key $\mathbf{k}L_i$ does not exist in R, and 0 otherwise.

Towards that end, consider invoking FromLtoR$([[\sigma]], [[1^m]])$, where σ is the sorting permutation used above and 1^m is the all one vector of size m. We now view the intermediate values: $\vec{f}, \vec{g}, \vec{h}$:

$\vec{f}$ =

⟦1⟧	⟦1⟧	⟦1⟧	(0)	(0)	(0)	(0)	[-1]⟧	[-1]⟧	[-1]⟧
L3	L5	L9	R3	R7	R9	R9	-L3	-L5	-L9

$\vec{g} = (\texttt{ApplyPerm}, [[\vec{\sigma}]], [[\vec{f}]])$

$\vec{g}$ =

⟦1⟧	(0)	[-1]⟧	⟦1⟧	[-1]⟧	(0)	⟦1⟧	(0)	(0)	[-1]⟧
L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

$\vec{h} = \texttt{PrefixSum}(\vec{g})$

$\vec{h}$ =

⟦1⟧	(1)	[0]⟧	⟦1⟧	[0]⟧	(0)	⟦1⟧	(1)	(1)	[0]⟧
L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

Here, each R elements is 1 if and only if it has the same key as an element in L.

At this point, we perform one more step: each element in $\vec{h}$ receives the negation of the element on its immediate right neighbor (and for completeness, the last element remains the same). That is, e.g., the element associated with $-L3$ receives the negation of L5, and the element associated with L5 receives the negation of -L5. We get the following vector:

$\vec{p}$ =

⟦0⟧	(1)	[0]⟧	⟦1⟧	[1]⟧	(0)	⟦0⟧	(0)	(1)	[0]⟧
L3	R3	-L3	L5	-L5	R7	L9	R9	R9	-L9

We now apply the inverse permutation to return the elements to their original position, i.e., run $(\texttt{ApplyInv}, [[\vec{\sigma}]], [[\vec{p}]])$ to get

⟦0⟧	⟦1⟧	⟦0⟧	(1)	(0)	(0)	(1)	[0]⟧	[1]⟧	[0]⟧
L3	L5	L9	R3	R7	R9	R9	-L3	-L5	-L9

Now we can return the first n elements, i.e., $([\![0]\!], [\![1]\!], [\![0]\!])$. Indeed, keys 3 and 9 appear in both tables in Figure 1, whereas 5 appears only in L. To see why this works, consider again the three sub-sequences that are in $\vec{g}$:

(1) **Type I: sub-sequence of the form** $\boxed{1}, (0), \ldots, (0), \boxed{-1}$**:** Recall that this sub-case occurs if there exists a key $\mathbf{k}L_j$ in L that appear in R, and thus the values that are associated with R will become (1) after the prefix sum. In that case, we want the indicator bit to be 0, i.e., that $\boxed{1}$ will become $\boxed{0}$. We obtain exactly that by receiving the negation of the immediate right neighbor.
(2) **Type II: sub-sequence of the form** $\boxed{1}, \boxed{-1}$**.** This sub-sequence occurs if the key $\mathbf{k}L_j$ does not appear in R. In that case, in $\vec{h}$ we will have $(\boxed{1}, \boxed{0})$, and we want that the indicator bit would be 1. I.e., we want that $\boxed{1}$ will remain $\boxed{1}$. By receiving the negation of the immediate right neighbor, this $\boxed{1}$ stays $\boxed{1}$.
(3) **Type III: sub-sequence of the form** $(0), \ldots, (0)$**:** This sub-sequence does not involve elements from L, so it is not relevant.

After "unsorting" the elements return to their original place, and the first n elements are exactly the indicator bits. We proceed with the pseudo-code of the procedure in Protocol 3.5.

Protocol 3.5 ($[[\vec{c}]] \leftarrow$ OnlyInL($[[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]]$))**:**

Input: (Shares of) the keys $[[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]]$, i. .e., the keys of the left table and the right table, respectively.
Output: (Shares of) a vector $[[\vec{c}]] = ([[c_1]], \ldots, [[c_m]])$, where $c_i = 1$ if the key $\mathbf{k}L_i$ does not exist in R, and $c_i = 0$ otherwise.
The protocol:
(1) Run (GenPerm, $[[\vec{\mathbf{k}L}]], [[\vec{\mathbf{k}R}]], [[\vec{\mathbf{k}L}]]$) by calling $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{\sigma}]]$.
(2) The parties generate n shares of $[[0]]$ (via calls to $\mathcal{F}_{\text{zero}}$).
(3) The parties generate $2m$ shares of $[[1]]$ (via calls to $\mathcal{F}_{\text{one}}$).
(4) Consider the vector $[[\vec{f}]]$ defined as $([[\vec{1^m}]], [[\vec{1^n}]], [[\vec{-1^m}]])$.
(5) The parties send (ApplyPerm, $[[\vec{\sigma}]], [[\vec{f}]]$) to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{g}]]$.
(6) $[[\vec{h}]] \leftarrow$ PrefixSum($[[\vec{g}]]$) (see Section 2).
(7) For every $i \in [2m + n - 1]$: $[[p_i]] = 1 - [[g_i]]$.
(8) Let $[[p_{2m+n}]] = [[g_{2m+n}]]$ and $\vec{p} = (p_1, \ldots, p_{2m+n})$.
(9) Run $\vec{f'} =$ (ApplyInv, $[[\vec{\sigma}]], [[\vec{p}]]$) by calling $\mathcal{F}_{\text{sort}}$.
(10) **Output:** $(f'_1, \ldots, f'_m)$, i.e., only the first m values of $\vec{f'}$.

Implementation note: In the inner join protocol (Protocol 3.3), in Step 2b, we already invoke FromLtoR on the all one vector, and therefore we already obtained Step 1-6 in OnlyInL. Therefore, we re-use these values and do not re-compute them. Consequently, we divided the protocol into corresponding different procedures for modularity and readability.

3.7 Left Outer Join

We provide the left outer join in Protocol 3.6. In the protocol, the parties first compute the inner join. Then, they compute the indicator bits of which keys in L do not appear in R using Protocol 3.5 and select those rows from L using $\mathcal{F}_{\text{mult}}$.

3.8 Full Join

Our full join protocol can be obtained by replacing InnerJoin in LeftOuterJoin (Protocol 3.6) with RightOuterJoin. Therefore, we omit the details.

Protocol 3.6 ($[[J]] \leftarrow$ LeftOuterJoin($[[L]], [[R]]$))**:**

Input: (Shares of) tables L and R shown in Fig. 3 (both tables are assumed to be sorted according to their keys).
Output: Table as defined in Definition 3.1 for the outer join operation.
The protocol:
(1) $[[J']] \leftarrow$ InnerJoin($[[L]], [[R]]$), i.e., Protocol 3.3.
(2) Let $[[\vec{\sigma}]]$ be the shares computed in line 1 of InnerJoin.
(3) Invoke $[[\vec{c}]] =$ OnlyInL($[[\vec{\sigma}]]$), where $\vec{c} = (c_1, \ldots, c_m)$ such that $c_i = 1$ iff the key $\mathbf{k}L_i$ does not exist in R.
(4) Let J be $(n + m) \times (d + e + 2)$ matrix where the first n rows are J'. We now fill the next m rows.
(5) For every $i \in [m]$:
  (a) Set the key as $[[\mathbf{k}J_{n+i}]] = [[\mathbf{k}L_i]]$.
  (b) For every $j \in [d]$: The parties send $([[c_i]], [[L_{i,j}]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[JL_{n+i,j}]]$.
  (c) The parties generate e shares of 0 by calling $\mathcal{F}_{\text{zero}}$ and set $[[JR_{i,j}]] = [[0]]$ for every $1 \le j \le e$.
  (d) The parties send $([[c_i]], [[\mathbf{v}L_i]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[\mathbf{v}J_{n+i}]]$.
(6) **Output:** J.

3.9 Efficiency and Security Analysis

We next prove the security of our protocol. Since the protocols are very similar, we analyze only the inner join protocol (Protocol 3.3). Let $\mathcal{F}_{\text{eqjoin}}$ be an ideal functionality that receives from the honest parties shares of the two tables, reconstructs them, and then hands the corrupted parties shares of the inner join of the two tables as defined in Definition 3.1. If the adversary is malicious, $\mathcal{F}_{\text{eqjoin}}$ first sends the adversary its shares of the input (as defined from the inputs of the honest parties) and allows the adversary to choose the corrupted parties' shares and define the shares of the honest parties accordingly. Then, we prove the following:

Theorem 3.7. *Protocol 3.3 securely computes $\mathcal{F}_{eqjoin}$ with abort in the $(\mathcal{F}_{zero}, \mathcal{F}_{one}, \mathcal{F}_{mult}, \mathcal{F}_{sort})$-hybrid model against malicious adversaries controlling any $t < n/2$ parties.*

Proof: Let $\mathcal{S}$ be the ideal world simulator and $\mathcal{A}$ be the real-world adversary. In the simulation, $\mathcal{S}$ interacts with $\mathcal{A}$ and plays the role of the ideal functionalities $\mathcal{F}_{\text{zero}}, \mathcal{F}_{\text{one}}, \mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{sort}}$. Observe that in the protocol, the parties do not interact at all but only hand inputs to the ideal functionalities to receive back an output. Also note that in the definition of all three ideal functionalities, the adversary receives its input shares, and then it can choose its output shares. The simulator, therefore, first receives the shares of L and R from the $\mathcal{F}_{\text{eqjoin}}$. The entire computation then proceeds as follows: The adversary receives its shares from the simulated ideal functionality $(\mathcal{F}_{\text{zero}}, \mathcal{F}_{\text{one}}, \mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{sort}})$ and chooses its output shares. In between invocations of ideal functionalities, it just performs some local computations, and their result is supposed to be the input for the following ideal invocation. As a result, the simulator (which knows the output of the adversary in the previous ideal execution) can perform the local computation on the shares and hand the adversary its shares in the following ideal invocation. It is immediate that the view of the adversary during the simulation is identically distributed to its view in the real-world execution. □

	$\mathcal{F}_{\text{mult}}$	$\mathcal{F}_{\text{sort}}$	
		GenPerm	ApplyPerm
Right outer-join	-	1	2
Inner join	$n \cdot e + d$	1	4
Left outer join	$n \cdot e + m(d+1) + d$	1	5
Full join	$m \cdot e$	1	3

Table 3: Cost of our different join protocols, measured by the number of calls to the multiplication and sorting functionalities. Recall that $m \times d$ and $n \times e$ are the dimension of the left table and right table, respectively. Note that $\mathcal{F}_{\text{sort}}$ has two operations: generating a permutation and applying it. The cost of each operation depends on the size of the input vector, which in our protocols is always $2m + n$.

Cost analysis. In all our protocols, the only interaction between the parties is inside the ideal functionalities $\mathcal{F}_{\text{sort}}$, $\mathcal{F}_{\text{zero}}$, $\mathcal{F}_{\text{one}}$, and $\mathcal{F}_{\text{mult}}$. As explained in Section 2, $\mathcal{F}_{\text{zero}}$ and $\mathcal{F}_{\text{one}}$ can be realized without interaction for a small number of parties. Hence, we analyze the cost of our protocols in terms of the number of calls to $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{sort}}$; see Table 3.

3.10 Join-uu

We can simplify the above protocols to the join-uu setting, where both tables contain only unique keys. Our join-uu protocol is based on the join-un protocol and is reduced to sorting and circuit evaluation.

The difference between join-uu and join-un is the method of copying the value which is part of the join, i.e., FromLtoR protocol. We show the protocol for the uu setting, FromLtoR-uu, in Protocol 3.9 and inner join protocol using FromLtoR-uu in Protocol 3.8.

In the join-un protocol, if there is a duplicate key with a certain value v, we compute $([[L_i]], [[0]], \ldots, [[0]], [[-L_i]])$ in FromLtoR (Algorithm 3.4) and use prefix-sum to copy L_i to 0s. On the other hand, in the case of join-uu, the keys are unique, so at most, only a single 0 follows L_i. This simplifies sandwiching 0 by L_i and $-L_i$. More specifically, in Step 1 in join-un, a permutation is generated from $[[\vec{\mathbf{k}}L]]||[[\vec{\mathbf{k}}R]]||[[\vec{\mathbf{k}}L]]$, but in join-uu the permutation can be generated from $[[\vec{\mathbf{k}}L]]||[[\vec{\mathbf{k}}R]]$. Then, a flag is generated indicating whether the key matches the next record's, and the previous value is copied accordingly using the Ifthen procedure. This reduces the input size for GenPerm from $2m + n$ to $m + n$. We remark that GenPerm is the heaviest part of the sorting, and this of course has a direct corresponding to the number of elements being sorted. The difference between join-un and join-uu is the FromLtoR protocol.

We remark that in the setting of join-uu, it is easy to extend the inner join protocol to left/right/full outer joins as claimed in [23].

3.11 Removing Null Rows

In all protocols above, the results include Null rows. Here, we show how to remove those null rows. As we mentioned above, removing the rows reveal the size of the intersection. Therefore, whether or not to remove the null rows depends on the application.

Our protocol. We describe our protocol that removes invalid rows in Protocol 3.10. The protocol computes the permutation that sorts the "valid bit" column while preferring valid rows over invalid. Then, it applies this permutation to each column separately, which moves all the valid rows to the beginning of the table. Then, the parties compute the shares of the number of valid rows, num, simply by summing all elements in the valid column. The parties reveal num by publicly reconstructing it, and the parties output the first

Protocol 3.8 ($[[J]] \leftarrow$ InnerJoin($[[L]], [[R]]$)):

Input: (Shares of) tables $[[L]]$ and $[[R]]$ shown in Fig. 3 (both tables are assumed to be sorted according to their keys).
Output: (Shares of) Table $[[J]]$ as defined in Definition 3.1 for the inner-join operation.
The protocol:

(1) For every $k \in [d]$, The parties invoke $[[J\vec{L}_k]] = ([[JL_{k,1}]], \ldots, [[JL_{k,n}]]) =$ FromLtoR-uu($[[\vec{\mathbf{k}}L]], [[\vec{\mathbf{k}}R]], [[\vec{L}_k]]$), where L_k denotes the kth column of L (see Section 3.2).
(2) **Computing $JR_1, \ldots, JR_e$ columns**:
  (a) Generate a vector $[[1]]^m$ by invoking $\mathcal{F}_{\text{one}}$ m times.
  (b) Invoke $[[\vec{c}]] =$ FromLtoR-uu($[[\vec{\mathbf{k}}L]], [[\vec{\mathbf{k}}R]], [[1]]^m$). Note that $c_i = 1$ iff there exists j such that $\mathbf{k}L_j = \mathbf{k}R_j$.
  (c) For every $i \in [n]$ and $j \in [e]$: the parties send $([[c_i]], [[R_{i,j}]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[JR_{i,j}]]$.
(3) **Computing $\mathbf{v}J$ column:**
  (a) Invoke $[[\vec{v}]] =$ FromLtoR-uu($[[\vec{\mathbf{k}}L]], [[\vec{\mathbf{k}}R]], [[\vec{\mathbf{v}}L]]$), where $v_i = 1$ if there exists j such that $\mathbf{k}L_j = \mathbf{k}R_i$ and $\mathbf{v}L_j = 1$.
  (b) for every $1 \le i \le d$, send $([[v_i]], [[\mathbf{v}R_i]])$ to $\mathcal{F}_{\text{mult}}$ and receive $[[\mathbf{v}J_i]]$.
(4) **Output:** $J =$

$$\{([[\mathbf{k}R_i]], [[JL_{i,1}]], \ldots, [[JL_{i,d}]], [[JR_{i,1}]], \ldots, [[JR_{i,e}]], [[\mathbf{v}J_i]])\}_{1 \le i \le n}.$$

Protocol 3.9 ($[[\vec{x}]] \leftarrow$ FromLtoR-uu($[[\vec{\mathbf{k}}L]], [[\vec{\mathbf{k}}R]], [[\vec{A}]]$)):

Input: (Shares of) the keys $[[\vec{\mathbf{k}}L]], [[\vec{\mathbf{k}}R]]$, i.e., the keys of the left table and the right table, respectively. In addition, shares of a vector $A = (A_1, \ldots, A_m)$.
Output: (Shares of) a vector $[[\vec{x}]] = ([[x_1]], \ldots, [[x_n]])$, where $x_i = A_i$ if there exists $\mathbf{k}L_j = \mathbf{k}R_i$, and 0 otherwise.
The protocol:

(1) The parties send (GenPerm, $([[\vec{\mathbf{k}}L]] \parallel [[\vec{\mathbf{k}}R]])$) to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{\sigma}]]$.
(2) The parties send (ApplyPerm, $([[\vec{\mathbf{k}}L]] \parallel [[\vec{\mathbf{k}}R]])$) to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{\mathbf{k}}X]]$.
(3) The parties generate flags $[[\vec{e}]]$ such that $e_i = 1$ if $\mathbf{k}X_i = \mathbf{k}X_{i+1}$ which means the keys are matched: for every $i \in [n + m - 1]$, $[[e_i]] =$ Mod2ToModP(Equal($[[\mathbf{k}X_i]], [[\mathbf{k}X_{i+1}]]$)).
(4) The parties generate n shares of $[[0]]$ (via calls to $\mathcal{F}_{\text{zero}}$). Recall that n is the number of rows in the table R.
(5) Consider the vector $[[\vec{f}]]$ defined as
$$([[A_1]], \ldots, [[A_m]], [[0]]^n).$$
(6) The parties send (ApplyPerm, $[[\vec{\sigma}]], [[\vec{f}]]$) to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{g}]]$.
(7) For every $i \in [n + m - 1]$, $[[h_{i+1}]] \leftarrow$ Ifthen($[[e_i]]$: $[[g_i]], [[0]]$). Let $[[h_1]] = [[0]]$.
(8) The parties send (ApplyInv, $[[\vec{\sigma}]], [[\vec{h}]]$) to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{f'}]] = ([[f'_1]], \ldots, [[f'_{m+n}]])$. f
(9) Output: $[[\vec{x}]] := ([[f'_{m+1}]], \ldots, [[f'_{m+n}]])$.

num rows. Note that there is an alternative way to replace the sorting by shuffling. We show that way in Section 3.11.

Protocol 3.10 ($[[J']] \leftarrow$ RemoveNullRows($[[J]]$)**:**

Input: (Shares of) table J as in Fig. 3. To ease notation, J contains A rows and B columns. The last column of J (the Bth column) contains validity bits.

Output: (Shares of) table J' which is obtained by removing invalid rows (i.e. removing all rows $a \in [A]$ with $J[a, B] = 0$) from J.

The protocol:

(1) The parties send $(\texttt{GenPerm}, [[\vec{J}_B]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) preferring 1 over 0, where J_B denotes the last column of J (which contains the validity bits). The parties receive $[[\vec{\tau}]]$.

(2) For every $j \in [B]$: The parties call to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) $(\texttt{ApplyPerm}, [[\vec{\tau}]], ([[J_{1,j}]], \ldots, [[J_{A,j}]]))$ and receive back the shares $([[J'_{1,j}]], \ldots, [[J'_{A,j}]])$.

(3) $[[\mathsf{num}]] := \sum_{i=1}^{A} [[J_{i,B}]]$, i.e., share of the number of valid rows.

(4) The parties reveal num by running Reconstruct($[[\mathsf{num}]]$), see Section 2.

(5) $[[J'_{i,j}]]_{1 \le i \le \mathsf{num}, j \in [B]}$.

Security. Let $\mathcal{F}_{\text{RemoveNull}}$ be an ideal functionality that receives shares of J from the honest parties, reconstructs the table, and hands the parties shares of J' while allowing the adversary to choose the corrupted parties' shares. Unlike the previous protocols, where the parties do not interact (but only communicate with the ideal functionalities), here the parties interact to reveal num. For proving security, we must allow the ideal adversary to learn num before handing the output to the honest parties. Hence, we let $\mathcal{F}_{\text{RemoveNull}}$ hand num to the adversary. Note that this is not private information and is learned by the adversary in the protocol anyway.

Theorem 3.11. *Protocol 3.10 securely computes $\mathcal{F}_{\text{RemoveNull}}$ with abort in the $\mathcal{F}_{\text{sort}}$-hybrid model against malicious adversaries controlling any $t < n/2$ parties.*

Proof: Let $\mathcal{S}$ be the ideal world simulator and $\mathcal{A}$ be the real-world adversary. The simulation of the ideal functionality $\mathcal{F}_{\text{sort}}$ is trivial since it only receives the output shares from $\mathcal{A}$ without outputting anything. To simulate the opening of num, we note that (i) $\mathcal{S}$ knows the corrupted parties' shares of num. This follows since these are computed by performing linear operations over the output shares of the ApplyPerm command, which are known to $\mathcal{S}$. Thus, the simulator $\mathcal{S}$ can carry-out the local linear operations over the shares it holds, and obtain the corrupted parties' shares. (ii) $\mathcal{S}$ knows num. This follows since before the opening of num, it receives num from the trusted party computing $\mathcal{F}_{\text{RemoveNull}}$. Thus, to simulate the opening of num, $\mathcal{S}$ chooses the honest parties' shares randomly under the constraint that together with the corrupted parties' shares, they will reconstruct to num. Then, it sends them to $\mathcal{A}$, simulating the honest parties sending their shares to the corrupted parties in the Reconstruct procedure. Upon receiving back the corrupted parties' shares, if an honest party P_j aborts the computation, since it cannot reconstruct num, then $\mathcal{S}$ sends abort_j to the trusted party computing $\mathcal{F}_{\text{RemoveNull}}$. Otherwise, it sends continue_j to the trusted party. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

The only difference between the simulation and a real execution is how the honest parties' shares of num are computed. However, since in both executions the shares are random and reconstructed to the same num, given the corrupted parties' shares, the distribution is the same. □

Cost of removing nulls. The cost of the protocols is one call to GenPerm and one call to ApplyPerm. The cost of each instruction depends on the size of the sent input table. The parties also run the Reconstruct procedure, but the cost of this is amortized away.

Alternative Way To Remove Null Rows We describe an alternative protocol for removing null rows in Protocol 3.12. In protocol 3.10, the rows with a valid bit of 0 are moved backward and deleted by outputting the first num rows. On the other hand, the alternative way is shuffling rows, revealing the valid bit, and extracting rows with a valid bit is 1. The revealed value is not num but 0 or 1; however, the random shuffling makes the order uniformly random so the revealed information for an adversary is the same as revealing num, and the same security statement is satisfied.

This method is more efficient than protocol 3.10 if the share generation of random permutation is more efficient than GenPerm of valid bit. In the three-party case, random permutation can be generated without communication as shown in [5], and this alternative way is therefore efficient.

Protocol 3.12 ($[[J']] \leftarrow$ RemoveNullRows2($[[J]]$)**:**

Input: (Shares of) table J as in Fig. 3. To ease notation, J contains A rows and B columns. The last column of J (the Bth column) contains validity bits.

Output: (Shares of) table J' which is obtained by removing invalid rows (i.e. removing all rows $a \in [A]$ with $J[a, B] = 0$) from J.

The protocol:

(1) The parties locally generate shares $[[\vec{\tau}]]$ of a random permutation.

(2) For every $j \in [B]$: The parties call to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) $(\texttt{ApplyPerm}, [[\vec{\tau}]], ([[J_{1,j}]], \ldots, [[J_{A,j}]]))$ and receive back the shares $([[J'_{1,j}]], \ldots, [[J'_{A,j}]])$.

(3) For $i \in [A]$, the parties reveal $J'_{i,B}$ by running Reconstruct($[[J'_{i,B}]]$), see Section 2.

(4) $[[J'_{i,j}]]_{i:J'_{i,B}=1, j \in [B]}$.

4 SECURE GROUP-BY PROTOCOL

In this section, we present our secure group-by protocols. Recall from the introduction that we differentiate between three different orders of the table we analyze: (1) *Input order* (the keys appear in arbitrary manner); (2) *grouped order* (the keys are sorted so rows with the same keys are located one next to another, it is easier to compute aggregating functions); (3) and *output order*, where each key appears only once, and the "value" corresponds to the aggregated information on the entire group. See also Figure 2.

There are several variants according to which aggregation function is applied to the group. Let T denote the input table containing m rows of pairs $\langle key, value\rangle$, and let $K = \{k_1, \ldots, k_n\}$ be the set of unique keys (ordered in ascending order). For a key $k \in K$, we let $\mathsf{Val}[k]$ be the set of values that are associated with k, that is, $\mathsf{Val}[k] = \{v \mid (k, v) \in T\}$. We list the different variants of group-by that we support. Each operation outputs a table consisting of the key of each group together with the aggregate function on $\mathsf{Val}[k]$, padded with $m - n$ rows of null.

Operation	Aggregation Function
GroupBy.count	$\lvert\mathsf{Val}[k]\rvert$
GroupBy.sum	$\sum_{v\in\mathsf{Val}[k]} v$
GroupBy.mean	$(\sum_{v\in\mathsf{Val}[k_i]} v)/\lvert\mathsf{Val}[k_i]\rvert$
GroupBy.max	$\max_v\{\mathsf{Val}[k_i]\}$
GroupBy.min	$\min_v\{\mathsf{Val}[k_i]\}$
GroupBy.median	$\mathsf{Median}(\mathsf{Val}[k_i])$

The ideal functionality. We define the general functionality $\mathcal{F}_{\mathrm{GB.X}}$ parameterized by a function $X \in \{\text{count}, \text{sum}, \text{mean}, \text{max}, \text{min}, \text{median}\}$. The functionality receives as input shares of $[[\vec{k}]]$ and $[[\vec{v}]]$ from all honest parties. The functionality then reconstructs $\vec{k}$ and $\vec{v}$, and hands the adversary its shares. It then sorts the keys and values into sets, finds the set of unique keys $K = \{k_1, \ldots, k_n\}$ and defines values $\mathsf{Val}[k_i]$ for each key k_i. Then, it defines the table T as $(k_i, X(\mathsf{Val}[k_i]))_{i=1}^{n}$, followed by $m - n$ rows of $(\mathtt{null}, 0)$. It receives from the adversary its output shares, and generates the shares of the honest parties accordingly.

4.1 GroupBy.common: Common Process for Group-By Operations

Since the different group-by operations are similar in nature, we first introduce a common procedure that outputs intermediate data for the various group-by operations. The input to the procedure is the vector of keys $[[\vec{k}]]$ and the vector of values $[[\vec{v}]]$. The output is as follows:

(1) $[[\vec{k_G}]]$: Shares of the keys $\vec{k}$ in a grouped-order.
(2) $[[\vec{v_G}]]$: Shares of the values $\vec{v}$ in a grouped-order.
(3) $[[\vec{e}]]$: Shares of flags, where $e_i = 1$ if $k_G[i]$ is the last element in its group, and 0 otherwise.
(4) $[[\vec{k_{GN}}]]$: the same order of $\vec{k_G}$ such that only the last element in each group is not null (and thus the keys are unique). That is, $k_{GN}[i] = \begin{cases} k_G[i] & \text{if } e_i = 1 \\ \mathtt{null} & \text{otherwise} \end{cases}$.
(5) $[[\pi_{\mathrm{GNtoOUT}}]]$: A permutation that sorts k_{GN} while moving null values to the end. If applied on $\vec{k_{GN}}$, it computes the output order of the keys.
(6) $[[\vec{k_{OUT}}]]$: shares of the keys in the output order (padded with nulls).

We demonstrate the output of GroupBy.common on the input of Figure 2. The protocol for GroupBy.common is described in Protocol 4.1.

Input		Output					
$\vec{k}$	$\vec{v}$	$\vec{k_G}$	$\vec{v_G}$	$\vec{e}$	$\vec{k_{GN}}$	$\pi_{\mathrm{GN}\vec{\mathrm{to}}\mathrm{OUT}}$	$\vec{k_{OUT}}$
1	2	1	2	0	null	4	1
3	4	1	3	1	1	1	2
1	3	2	1	1	2	2	3
3	5	3	4	0	null	5	null
2	1	3	5	1	3	3	null
Input O.		Group O.					Output O.

A note about sorting: In Step 1 the protocol invokes the GenPerm protocol to compute a permutation which sorts the inputs based on both k and v. (Namely, if two rows have the same key and different values, the row with the smaller value will be located first.) Using v as part of the sorting key is required for computing GroupBy.max, GroupBy.min and GroupBy.median, but is not required for computing GroupBy.count, GroupBy.sum and GroupBy.mean. Therefore in the latter case the protocol can compute GenPerm using only the k values. This can substantially improve the overhead, since the overhead of sorting depends on the length of the keys.

Protocol 4.1 $((\vec{[[k_G]]}, \vec{[[v_G]]}, [[\vec{e}]], [[\pi_{\mathrm{GN}\vec{\mathrm{to}}\mathrm{OUT}}]], \vec{[[k_{GN}]]}, \vec{[[k_{OUT}]]}) \leftarrow \text{GroupBy.common}([[\vec{k}]], [[\vec{v}]]))$**:**

Input: Keys $[[\vec{k}]]$ and values $[[\vec{v}]]$ of length m. The bit length of each key is ℓ-bit.
Output: see in the description above.
The protocol:

(1) The parties send $(\mathtt{GenPerm}, [[\vec{k}, \vec{v}]])$[2] to $\mathcal{F}_{\mathrm{sort}}$ (Functionality 2.1) and receive $[[\vec{\pi}]]$.
(2) The parties invoke $\mathcal{F}_{\mathrm{sort}}$ (Functionality 2.1) twice with inputs $(\mathtt{ApplyPerm}, [[\vec{\pi}]], ([[\vec{k}]]))$ and $(\mathtt{ApplyPerm}, [[\vec{\pi}]], ([[\vec{v}]]))$ to receive $\vec{[[k_G]]}$, $\vec{[[v_G]]}$.
(3) $\forall i \in [m-1]$, set $[[f_i]] \leftarrow \text{equall}([[k_G[i]]], [[k_G[i+1]]])$, and $[[e_i]] = 1 - [[f_i]]$.
(4) Set $[[e_m]] = [[1]]$ (by calling $\mathcal{F}_{\mathrm{one}}$).[3]
(5) $\forall i \in [m]$, set $[[k_{GN}[i]]] \leftarrow \text{Ifthen}([[e_i]] : [[k_G[i]]], [[\mathtt{null}]])$.
(6) The parties send $(\mathtt{GenPerm}, [[\vec{e}]])$ to $\mathcal{F}_{\mathrm{sort}}$ (Functionality 2.1) with an order preferring 1 over 0, and receive $[[\pi_{\mathrm{GN}\vec{\mathrm{to}}\mathrm{OUT}}]]$.
(7) The parties send $(\mathtt{ApplyPerm}, [[\pi_{\mathrm{GN}\vec{\mathrm{to}}\mathrm{OUT}}]], \vec{[[k_{GN}]]})$ to $\mathcal{F}_{\mathrm{sort}}$ (Functionality 2.1) and receive $\vec{[[k_{OUT}]]}$.
(8) Output $(\vec{[[k_G]]}, \vec{[[v_G]]}, [[\vec{e}]], \vec{[[k_{GN}]]}, [[\pi_{\mathrm{GN}\vec{\mathrm{to}}\mathrm{OUT}}]], \vec{[[k_{OUT}]]})$.

4.2 Group-by-Count

We show GroupBy.count in Protocol 4.2, followed by an example. We use GroupBy.common and look at the grouped order. We give each element with $e_i = 1$ its rank in the grouped order list. We then

[2] We sort keys and values together such that, within each group, we prefer values with smaller values over larger values.
[3] We use here a generation of shares of 1. If the threshold is tight, i.e., $n = 2t + 1$, then this is not needed, and we can just take the constant 1. However, we prove our protocol for *any* $t < n/2$. If the threshold used for the secret sharing scheme is higher than the actual number of corruptions, then the shares of the honest parties should remain secret even though the secret is known.

move to the output order by applying π_{GNtoOUT}, and compute the differences of the ranks of adjacent elements in this order. Recall that if we compute only GroupBy.count, we skip the processes regarding values $[[\vec{v}]]$ in GroupBy.common because GroupBy.count does not change the values. We, therefore, abuse notation and use only the output we actually need from GroupBy.common, and the implementation we optimize it as well.

Protocol 4.2 $((k_{\text{OUT}}, [[\vec{c}]]) \leftarrow \text{GroupBy.count}([[\vec{k}]], [[\vec{v}]]))$**:**

Input: Keys $[[\vec{k}]]$ and values $[[\vec{v}]]$
The protocol:
(1) $([[\vec{k_G}]], [[\vec{e}]], [[\pi_{\vec{\text{GNtoOUT}}}]], [[\vec{k_{\text{OUT}}}]]) \leftarrow \text{GroupBy.common}([[\vec{k}]])$.
(2) For every $i \in [m]$: $[[x_i]] \leftarrow \text{IfThen}([[e_i]] : i, m)$.
(3) The parties send $(\texttt{ApplyPerm}, [[\pi_{\vec{\text{GNtoOUT}}}]], [[\vec{x}]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{y}]]$
(4) $[[c_1]] := [[y_1]]$
(5) For every $i \in [2, m]$: $[[c_i]] = [[y_i]] - [[y_{i-1}]]$
(6) Output $([[\vec{k_{\text{OUT}}}]], [[\vec{c}]])$

$\vec{k}$	$\vec{v}$	$\vec{k_G}$	$\vec{e}$	$\vec{x}$	$\vec{k_{\text{OUT}}}$	$\vec{y}$	$\vec{c}$
1	2	1	0	5	1	2	2
3	4	1	1	2	2	3	1
1	3	2	1	3	3	5	2
3	5	3	0	5	null	5	0
2	1	3	1	5	null	5	0
Input O.		Grouped O.			Output O.		

Table 4: Example for group-by-count.

We next show an example GroupBy.count in Table 4. To see why the protocol is correct, in the vector $\vec{x}$, we have that: $x_i = i$ if $k_G[i]$ is the last element in its group, and $x_i = m$ otherwise. In other words, if $k_G[i]$ is the last key in its group, then it contains the total number of elements with keys smaller or equal to the key. Applying π_{GNtoOUT} moves $\vec{x}$ from grouped order to output order, and so in $\vec{y}$, each element y_i that corresponds to a key that is not null, contains the total number of elements with keys smaller or equal to k_i. By subtracting $c_i = y_i - y_{i-1}$, we get exactly the number of elements with the key k_i. Moreover, the last element that does not correspond to null in y (i.e., the value of the last key) contains the total number of elements, m. Each element that corresponds to null received $x_i = m$, and thus by taking $y_i - y_{i-1}$ all the null values get $m - m = 0$.

Claim 4.3. *Protocol 4.2 securely computes the $\mathcal{F}_{\text{GroupBy.count}}$ functionality against malicious adversaries controlling any $t < n/2$ parties.*

Proof: To see why the protocol is correct, in the vector $\vec{x}$, we have that:

$$x_i = \begin{cases} i & \text{if } k_G[i] \text{ is the last element in its group} \\ m & \text{otherwise} \end{cases}.$$

In other words, if $k_G[i]$ is the last key in its group, then it contains the total number of elements with keys smaller or equal to the key. Applying π_{GNtoOUT} moves $\vec{x}$ from grouped-order to output-order, and so in $\vec{y}$, each element y_i that corresponds to a key that is not null, contains the total number of elements with keys smaller or equal to k_i. By subtracting $c_i = y_i - y_{i-1}$, we get exactly the number of elements with the key k_i. Moreover, the last element that does not correspond to null in y contains the total number of elements, m. Finally, each element that correspond to null received $x_i = m$, and thus by taking $y_i - y_{i-1}$ we get 0.

As for security, all operations are easy to simulate since we are working with shares and honest majority. Specifically, the simulator first receives shares from the ideal functionality of the corrupted parties. Then, all computations are local and deterministic, or that the parties receive some shares from some inner ideal functionality in the hybrid model (such as $\mathcal{F}_{\text{sort}}$, etc.). The simulator knows exactly the shares that the adversary is supposed to have as it holds the adversary's input shares and all computations are local. It then hands the adversary the shares it is supposed to receive from the inner functionality, and receives from the adversary its output shares in that functionality. It therefore can continue and compute locally the shares that the adversary is supposed to receive in the following ideal invocation. At the end of the execution, it can compute what the output shares of the adversary are supposed to be (those are just local computations over information that the simulator already has), and it hands those shares to $\mathcal{F}_{\text{GroupBy.count}}$. The trusted party then computes the shares of the honest parties given the shares of the corrupted parties.

It follows that the simulator perfectly simulates the view of the adversary. As such, the output of the adversary is exactly the same in both worlds. We also saw that the underlying secrets of the shares of the output are the same in both the real world and the ideal world. Thus, the shares of the honest parties are uniform conditioned on the shares of the adversary and the secrets. Therefore, the joint distribution of the view of the adversary and the shares of the honest parties is the same in the ideal-world and the real-world. □

4.3 Group-by-Sum and Mean

We show the group-by-sum protocol in Protocol 4.4. Note, if we compute only GroupBy.sum, we can remove values $[[\vec{v}]]$ from the sort keys in GroupBy.common because GroupBy.sum does not require the values $[[\vec{v_G}]]$ in a grouped-order to be sorted.

This protocol can also compute variants of the sum, such as the sum-of-squares, or an inner-product. This can be done by first running a protocol for computing the required multiplications and then executing the group-by-sum protocol.

$\vec{k}$	$\vec{v}$	$\vec{k_G}$	$\vec{e}$	$\vec{v_G}$	$\vec{w}$	$\vec{x}$	$\vec{k_{\text{OUT}}}$	$\vec{y}$	$\vec{s}$
1	2	1	0	2	2	15	1	5	5
3	4	1	1	3	5	5	2	6	1
1	3	2	1	1	6	6	3	15	9
3	5	3	0	4	10	15	null	15	0
2	1	3	1	5	15	15	null	15	0
Input O.		Grouped O.					Output O.		

Protocol 4.4 $([[\vec{k}_{\text{OUT}}]], [[\vec{s}]]) \leftarrow$ GroupBy.sum$([[\vec{k}]], [[\vec{v}]])$:

Input: Keys $[[\vec{k}]]$ and values $[[\vec{v}]]$.
Output: Keys in output order $[[\vec{k}_{\text{OUT}}]]$ and grouped sum $[[\vec{s}]]$
(1) $([[\vec{k}_{\text{G}}]], [[\vec{v}_{\text{G}}]], [[\vec{e}]], [[\pi_{\text{GNtoOUT}}]], [[\vec{k}_{\text{GN}}]], [[\vec{k}_{\text{OUT}}]]) \leftarrow$ GroupBy.common$([[\vec{k}]], [[\vec{v}]])$.
(2) $[[\vec{w}]] \leftarrow \text{PrefixSum}([[\vec{v}_{\text{G}}]])$
(3) For every $i \in [m]$: $[[x_i]] \leftarrow$ Ifthen$([[e_i]] : [[w_i]], [[w_m]])$.
(4) The parties send $(\texttt{ApplyPerm}, [[\pi_{\text{GNtoOUT}}]], [[\vec{x}]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{y}]]$
(5) For every $i \in [m]$: $[[s_i]] = [[y_i]] - [[y_{i-1}]]$, where $[[x'_0]] := [[0]]$
(6) Output $([[\vec{k}_{\text{OUT}}]], [[\vec{s}]])$

Claim 4.5. *Protocol 4.4 securely computes the $\mathcal{F}_{\text{GroupBy.sum}}$ functionality against malicious adversaries controlling any $t < n/2$ parties.*

Proof: We start with correctness. To see why the algorithm is correct, the vector $\vec{w}$ contains the prefix sum of the values in grouped-order. Namely, $w_i = \sum_{j \le i} v_{\text{G}}[j]$. In the vector $\vec{x}$, we have that x_i is:

$$x_i = \begin{cases} w_i \\ w_m \end{cases} = \begin{cases} \sum_{j \le i} v_{\text{G}}[j] & \text{if } k_{\text{G}}[i] \text{ is the last element in its group,} \\ \sum_{j=1}^{m} v_{\text{G}}[j] & \text{otherwise} \end{cases}.$$

Applying π_{GNtoOUT} on $\vec{x}$ moves all elements in the first branch to the top of the array. This is the vector $\vec{y}$. In the array $\vec{y}$, each key k_i contains the sum of all values with keys smaller or equal k_i. By subtracting $s_i = y_i - y_{i-1}$, we get exactly the sum of values of all keys with the key k_i. Moreover, rows corresponding to `null` values have the total sum of all values in the array. By computing $y_i - y_{i-1}$, we extract the sum of the keys of each group. Moreover, since all elements that correspond to `null` hold the same sum (the total sum), they all become 0s.

Security follows from the same reasoning as in Claim 4.3, and due to the fact that we have an honest majority. □

Cost comparison with a protocol in [17]. The protocol proposed as Algorithm 11 in [17] (called GroupSum2) is similar to our GroupBy.sum protocol, and GroupSum2 can also be used to compute group-by-sum. We compare the cost of computing group-by-sum using our protocol with that of using GroupSum2.

Assuming that GroupBy.common has already been executed, our GroupBy.sum requires one less call to `GenPerm` than GroupSum2 since GroupBy.sum is optimized for use after GroupBy.common but GroupSum2 is not. Our GroupBy.sum requires m calls to $\mathcal{F}_{\text{mult}}$ and one call to `ApplyPerm`, as shown in Table 6. On the other hand, the typical cost when using GroupSum2 is m calls to $\mathcal{F}_{\text{mult}}$, one call to `GenPerm`, and one call to `ApplyPerm`. This is achieved, for example, by the following computation.
(1) Locally compute the inputs of GroupSum2 from the outputs of GroupBy.common.
(2) Compute $[[\vec{s}_1]]$ by running Steps 1 through 4 of GroupSum2.
(3) Output $([[\vec{k}_{\text{OUT}}]], [[\vec{s}_1]])$ as $([[\vec{k}_{\text{OUT}}]], [[\vec{s}]])$.

If we do not assume the pre-execution of GroupBy.common, the number of calls to functionalities that require interaction is equal between the cases using GroupBy.sum and GroupSum2. When using GroupBy.sum, we sequentially perform GroupBy.common and GroupBy.sum, which requires $2m$ calls to $\mathcal{F}_{\text{mult}}$, two calls to `GenPerm`, four calls to `ApplyPerm`, and m calls to equality check. Note that we save one call to `GenPerm` by using $[[\vec{k}]]$ as the sort key instead of $[[\vec{k}, \vec{v}]]$ in Step 2 of GroupBy.common. When using GroupSum2, the natural protocol would be as follows.
(1) Run Steps 1 through 4 of GroupBy.common. We use $[[\vec{k}]]$ as the sort key instead of $[[\vec{k}, \vec{v}]]$.
(2) Locally compute the inputs of GroupSum2 from $[[\vec{v}_{\text{G}}]]$ and $[[\vec{e}]]$.
(3) Compute $[[\vec{s}_1]]$ by running Steps 1 through 4 of GroupSum2.
(4) Compute $[[\vec{k}_{\text{GN}}]]$ by running Step 5 of GroupBy.common using $[[\vec{g}]]$ instead of $[[\vec{e}]]$.
(5) Compute $[[\vec{k}_{\text{OUT}}]]$ by running Step 7 of GroupBy.common using $[[\pi]]$ computed in Step 2 of GroupSum2 instead of $[[\pi_{\text{GNtoOUT}}]]$.
(6) Output $([[\vec{k}_{\text{OUT}}]], [[\vec{s}_1]])$ as $([[\vec{k}_{\text{OUT}}]], [[\vec{s}]])$.
This protocol requires $2m$ calls to $\mathcal{F}_{\text{mult}}$, two calls to `GenPerm`, four calls to `ApplyPerm`, and m calls to equality check. This is the same cost as when using GroupBy.sum.

Group-by-Mean. Now that we have group-by-sum and count, we can also compute group-by-mean, since the mean is the sum divided by the count. When computing grouped statistics, it is common to reveal the count *and* sum (or mean), and in this case, the parties conduct group-by-sum and count, reveal them, and then obtain mean by dividing the sum by the count in plaintext. In the special case where the count is not revealed but the mean is to be computed, the parties divide the output of group-by-sum by that of group-by-count using division protocol (e.g., [6]).

4.4 Group-by-Min/Max

We show group-by-max and min protocols in Protocols 4.6 and 4.8.

Protocol 4.6 $(([[\vec{k}_{\text{OUT}}]], [[\vec{y}]]) \leftarrow$ GroupBy.max$([[\vec{k}]], [[\vec{v}]]))$:

Input: Keys $[[\vec{k}]]$ and values $[[\vec{v}]]$.
Output: Grouped keys $[[\vec{k}_{\text{OUT}}]]$ and its maximum $[[\vec{y}]]$
The protocol:
(1) $([[\vec{k}_{\text{G}}]], [[\vec{v}_{\text{G}}]], [[\vec{e}]], [[\pi_{\text{GNtoOUT}}]], [[\vec{k}_{\text{GN}}]], [[\vec{k}_{\text{OUT}}]]) \leftarrow$ GroupBy.common$([[\vec{k}]], [[\vec{v}]])$.
(2) For every $i \in [m]$: $[[x_i]] \leftarrow$ Ifthen$([[e_i]] : [[v_{\text{G}}[i]]], [[0]])$
(3) The parties send $(\texttt{ApplyPerm}, [[\pi_{\text{GNtoOUT}}]], [[\vec{x}]])$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $[[\vec{y}]]$
(4) Output $([[\vec{k}_{\text{OUT}}]], [[\vec{y}]])$

$\vec{k}$	$\vec{v}$	$\vec{k}_{\text{G}}$	$\vec{e}$	$\vec{v}_{\text{G}}$	$\vec{x}$	$\vec{k}_{\text{OUT}}$	$\vec{y}$
1	2	1	0	2	0	1	3
3	4	1	1	3	3	2	1
1	3	2	1	1	1	3	5
3	5	3	0	4	0	null	0
2	1	3	1	5	5	null	0
Input O.		Grouped O.				Output O.	

Claim 4.7. *Protocol 4.6 securely computes $\mathcal{F}_{\text{GroupBy.max}}$ against malicious adversaries controlling any $t < n/2$ parties.*

Proof: For correctness, recall GroupBy.common sorts the elements according to both the keys and the values. Therefore, the maximal

value within each group in v_G is the last element in each group. Those are exactly the locations where $e_i = 1$. Therefore, in $\vec{x}$, each x_i is either the maximal value in its group (if it is the last element in its group), or it is 0. By applying the permutation π_{GNtoOUT} on $\vec{x}$ we get exactly the output. Security follows from similar reasoning as in Claim 4.3. □

Min. We proceed with describing GroupBy.min in Protocol 4.8.

Protocol 4.8 $((\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{y} \rrbracket) \leftarrow \text{GroupBy.min}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket))$**:**

Input: Keys $\llbracket \vec{k} \rrbracket$ and values $\llbracket \vec{v} \rrbracket$.
Output: Grouped keys $\llbracket \vec{k}_{\text{OUT}} \rrbracket$ and its minimum $\llbracket \vec{y} \rrbracket$.
The protocol:

(1) $(\llbracket \vec{k}_G \rrbracket, \llbracket \vec{v}_G \rrbracket, \llbracket \vec{e} \rrbracket, \llbracket \pi_{\text{GNtoOUT}} \rrbracket, \llbracket \vec{k}_{GN} \rrbracket, \llbracket \vec{k}_{\text{OUT}} \rrbracket) \leftarrow$ GroupBy.common$(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket)$.
(2) For each $i \in [m]$:
  (a) $\llbracket x_i \rrbracket \leftarrow$ IfThen$(\llbracket e_{i-1} \rrbracket : \llbracket v_G[i] \rrbracket, \llbracket 0 \rrbracket)$ (where $\llbracket e_0 \rrbracket = \llbracket 1 \rrbracket$).
  (b) $\llbracket g_i \rrbracket := 1 - \llbracket e_{i-1} \rrbracket$
(3) The parties send $(\texttt{GenPerm}, \llbracket \vec{g} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket \pi_{\vec{\text{OUT}}} \rrbracket$.
(4) The parties send $(\texttt{ApplyPerm}, \llbracket \pi_{\vec{\text{OUT}}} \rrbracket, \llbracket \vec{x} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket \vec{y} \rrbracket$
(5) Output $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{y} \rrbracket)$

$\vec{k}$	$\vec{v}$	$\vec{k}_G$	$\vec{e}$	$\vec{v}_G$	$\vec{x}$	$\vec{g}$	$\vec{k}_{\text{OUT}}$	$\vec{y}$
1	2	1	0	2	2	0	1	2
3	4	1	1	3	0	1	2	1
1	3	2	1	1	1	0	3	4
3	5	3	0	4	4	0	null	0
2	1	3	1	5	0	1	null	0
Input O.				Grouped O.				Output O.

Claim 4.9. *Protocol 4.8 securely computes $\mathcal{F}_{\text{GroupBy.min}}$ against malicious adversaries controlling any $t < n/2$ parties.*

Proof: The structure of GroupBy.min is slightly different than our other group-by protocol, since we do not use π_{GNtoOUT}. To understand why group-by-min is correct, recall that after sorting the elements in GroupBy.common, the minimal value in each group is the first element (as we sort the keys and the values). Moreover $e_i = 1$ only if it is the last element in its group; this means that the first element in the next group is in index $i + 1$. This is why x_i relates to e_{i-1}. We compute vectors $\vec{x}$ and $\vec{g}$, satisfying:

$$x_i = \begin{cases} v_G[i] & \text{if } k_G[i] \text{ is the first element in its group} \\ 0 & \text{otherwise} \end{cases}.$$

and

$$g_i = \begin{cases} 0 & \text{if } k_G[i] \text{ is the first element in its group} \\ 1 & \text{otherwise} \end{cases}.$$

As such, $\vec{x}$ contains all the minimum values in each group, but it is not in an output-order. We also cannot use π_{GNtoOUT} since it moves the last element in each group to the top; not the first element in each group. We compute a permutation π_{OUT} which sorts $\llbracket \vec{g} \rrbracket$, and so it moves all rows with 0 to the beginning.

However, by the definition of $\vec{g}$, these rows correspond to the rows with the minimal value in each group. Thus, we can apply this permutation on $\vec{x}$, which brings all minimal values to the beginning of the output vector as required. □

4.5 Group-by-Median

In this section, we present a protocol for computing the median for each group. We use the following definition for the median: If the number of records is odd, the median is the middle value (in sorted order). In other words, it is the value for which the number of elements greater than it is equal to the number of elements smaller than it. If the number of records is even, the median is the average of the two middle records. For example, the median of the values $\{10, 20, 30, 40\}$ is $25 = (20+30)/2$. (We recognize that, in some cases, the median of a set with an even number of elements is defined differently. It is the value that is greater than exactly half of the elements, e.g., 30 in the example above. Our protocol can be adapted for computing this value as the median, as described in Footnote 4.)

In order not to leak whether the number of records is odd or even, the protocol computes "2 times the median," which is just 2 times the median if the number of records is odd and also the sum of the two middle values if the number of records is even. The original median can be obtained by dividing the reconstructed value by 2. If it is required to move the record to another protocol without reconstructing it, we can truncate the LSB by using a truncation protocol (e.g., from [6]).

The group-by-median uses three sub-protocols:

- rank$^+$ computes for each element its rank in its group, in ascending order and starting from 0. For example, if the same key appears in 4 rows, with the values $\{10, 20, 30, 40\}$, then the rank of 10 is 0, the rank of 20 is 1, etc. The output is given in a *grouped order.*
- rank$^-$ computes for each element its rank in its group in descending order. The output is given in a *grouped order.*
- FindZeroOrOne receives keys and values in a grouped order, together with a vector of values $\vec{x}$ with the guarantee that within each group, only one of the elements x_i has a value 0 or 1. The procedure selects only the values for which x_i is 0 or 1, and returns them in an output order.

Finding the median. After running rank$^+$ and rank$^-$, the protocol computes for each element the values `a-d`, which is equal to "ascending order rank - descending order rank" and `d-a` which is equal to "descending order rank - ascending order rank". If the number of records in a group is odd, then `a-d=d-a=` 0 for the median value. See, e.g., the keys 1 and 3 in Figure 4. If the number of records in a group is even, one of the two middle values has `a-d=` 1. The other middle value has `d-a=` 1. See, e.g., the key 2 in Figure 4. Note that in $a - d$, in each group, we have exactly one element that is 0 or 1; Likewise, in $d - a$, in each group, we have exactly one element that is 0 or 1. Therefore, we apply FindZeroOrOne on each separately, which returns exactly one value in each group and in output order. If the number of records in a group is odd, then in both executions, it would return the same value. If the number of records in a group is even, in each execution, we would get different values—the two middle values. We therefore sum the resulting

values and get either "2 times the median" (odd) or "the sum of two middle values" (even).[4]

We show the complete GroupBy.median in Protocol 4.10. Figure 4 shows the process example of GroupBy.median.

Protocol 4.10 $((\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{w} \rrbracket) \leftarrow \text{GroupBy.median}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket))$**:**

Input: Keys $\llbracket \vec{k} \rrbracket$ and values $\llbracket \vec{v} \rrbracket$.
Output: Grouped keys $\llbracket \vec{k}_{\text{G}} \rrbracket$ and its median $\llbracket \vec{w} \rrbracket$.

(1) Run $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{a} \rrbracket) \leftarrow \text{rank}^{+}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket)$.
(2) Run $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{d} \rrbracket) \leftarrow \text{rank}^{-}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket)$.
(3) Run $\llbracket \vec{z} \rrbracket \leftarrow \text{FindZeroOrOne}(\llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{v}_{\text{G}} \rrbracket, \llbracket \vec{a} - \vec{d} \rrbracket)$.
(4) Run $\llbracket \vec{w} \rrbracket \leftarrow \text{FindZeroOrOne}(\llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{v}_{\text{G}} \rrbracket, \llbracket \vec{d} - \vec{a} \rrbracket)$.
(5) $\llbracket \vec{v} \rrbracket = \llbracket \vec{w} \rrbracket + \llbracket \vec{z} \rrbracket$.
(6) Output: $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{v} \rrbracket)$.

$\vec{k}_{\text{G}}$	$\vec{v}_{\text{G}}$	$\vec{a}$	$\vec{d}$	$\vec{a}-\vec{d}$	$\vec{d}-\vec{a}$
1	3	0	2	-2	2
1	10	1	1	0	0
1	15	2	0	2	-2
2	2	0	1	-1	1
2	4	1	0	1	-1
3	1	0	0	0	0

Grouped Order

$\vec{k}_{\text{OUT}}$	$\vec{z}$	$\vec{w}$	$\vec{v}$
1	10	10	20
2	4	2	6
3	1	1	2
null	0	0	0
null	0	0	0
null	0	0	0

Output Order

Figure 4: Example for GroupBy.median protocol (Protocol 4.10). $\vec{a}$ is the result of rank^{+} (the rank of each element within its group in ascending order) and $\vec{d}$ is the result of rank^{-} (the rank of each element in its group in descending order). Recall that we compute "sum of medians" to hide the number of elements in each group.

Security. To prove the security of the protocol, we first formalize the functionalities $\mathcal{F}_{\text{rank}^{+}}$ (resp. $\mathcal{F}_{\text{rank}^{-}}$): The functionality receives the shares from the honest parties, reconstruct $\vec{k}$ and $\vec{v}$, and hands the adversary's its shares. It computes k_{OUT}, k_{G}. For computing $\vec{a}$, it just scans k_{G} and gives each key its rank within its group (in ascending order). Respectively, for computing $\vec{d}$ in rank^{-}, the functionality counts the number of elements in each group, say c_i for the ith group, and computes for the jth element in the group (starting from 0) the rank $c_i - j - 1$. In both functionalities, it asks the adversary for its shares and computes the shares of the honest parties based on the corrupted parties' shares and the secrets – k_{OUT}, k_{G}, and $\vec{a}$ (resp. $\vec{d}$).

We also formalize the ideal functionality of FindZeroOrOne, denoted as $\mathcal{F}_{\text{FindZeroOrOne}}$. The functionality receives from the honest parties shares of $\vec{k}_{\text{G}}$, $\vec{v}_{\text{G}}$ and $\vec{x}$. It verifies that for every group, exactly one element in each group has $x_i = 0$ or $x_i = 1$. If this condition does not hold, then the functionality aborts. The functionality then for each group k_i finds the elements v_j for which $x_j = 0$ or 1. It generates a table containing all these pairs, and pad it with $m - n$ null rows (where n is the number of unique keys). The interaction with the adversary is the same as in all other functionalities in our setting: When reconstructing the inputs it gives the corrupted parties their shares (which is determined from the shares of the honest parties). Before giving output to the honest parties, it first lets the adversary determine its output, and then the functionality chooses the shares of the honest parties accordingly.

Claim 4.11. *Protocol 4.10 securely computes the $\mathcal{F}_{\text{GroupBy.median}}$ functionality against malicious adversaries controlling any $t < n/2$ parties.*

Proof: Correctness holds by inspection: inside each group, if the number of elements is odd (say $2k+1$) then exactly one element in $\vec{a} - \vec{d}$ and in $\vec{d} - \vec{a}$ is 0. This is always the element with rank $k+1$, which is the median. FindZeroOrOne will find this element and will return it (twice).

If the number of elements is even (say $2k$) then exactly one element is 1 – this is the element with the rank $k+1$ in the group: its value in $\vec{a}$ is $k+1$ and in $\vec{d}$ is k. In $\vec{d} - \vec{a}$, there is exactly one element in $\vec{d} - \vec{a}$ that is 1: this is the element with rank k in the group: In $\vec{d}$ its value is $k+1$ and in $\vec{a}$ its value is k. Therefore, FindZeroOrOne finds the two elements with rank k and $k+1$ and the algorithm returns the sum of them.

Security follows from a similar reasoning as in Claim 4.3. □

4.6 Sub-protocols for Group-by-Median

We propose the three protocols rank^{+}, rank^{-}, FindZeroOrOne, which are sub-protocols for GroupBy.median.

rank^{+} The protocol for rank^{+} appears in Protocol 4.12.

Protocol 4.12 $((\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{a} \rrbracket) \leftarrow \text{rank}^{+}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket))$**:**

Input: Keys $\llbracket \vec{k} \rrbracket$ and values $\llbracket \vec{v} \rrbracket$
Output: Grouped keys $\llbracket \vec{k}_{\text{G}} \rrbracket$ and its ascending rank $\llbracket \vec{a} \rrbracket$ (starting rank is 0).
The protocol:
(1) $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{c} \rrbracket) \leftarrow \text{GroupBy.count}(\llbracket \vec{k} \rrbracket, \llbracket \vec{v} \rrbracket)$.
(2) Store $\llbracket \vec{k}_{\text{G}} \rrbracket$ and $\llbracket \pi_{\text{GNtoOUT}} \rrbracket$, which is outputs of GroupBy.common in the process of GroupBy.count.
(3) The parties send $(\texttt{ApplyInv}, \llbracket \pi_{\text{GNtoOUT}} \rrbracket, \llbracket \vec{c} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket \vec{d} \rrbracket$
(4) $\llbracket \vec{s} \rrbracket \leftarrow \text{PrefixSum}(\llbracket \vec{d} \rrbracket)$
(5) For every $i \in [m]$, $\llbracket a_i \rrbracket = (i-1) - \llbracket s_{i-1} \rrbracket$ (where $\llbracket s_0 \rrbracket = 0$). I.e., the current location $i-1$ minus the number of items in all preceding groups.
(6) Output $(\llbracket \vec{k}_{\text{OUT}} \rrbracket, \llbracket \vec{k}_{\text{G}} \rrbracket, \llbracket \vec{a} \rrbracket)$.

An example of an execution of Protocol 4.12 is given below.

[4] The protocol can easily be changed to compute the median according to the alternative definition, which sets the median of a set with an even number of items, 2ℓ, to be the item with rank $\ell + 1$. The adapted protocol only computes and examines a-d and applies FindZeroOrOne on that vector.

$\vec{k}$	$\vec{v}$	$\vec{k_{\text{OUT}}}$	$\vec{c}$	$\vec{k_{\text{G}}}$	$\vec{d}$	$\vec{s}$	$\vec{a}$
2	2	1	3	1	0	0	0
1	3	2	2	1	0	0	1
1	15	3	1	1	3	3	2
2	4	null	0	2	0	3	0
3	1	null	0	2	2	5	1
1	10	null	0	3	1	6	0
Input O.		Output O.		Grouped O.			

Claim 4.13. *Protocol 4.12 securely computes the $\mathcal{F}_{\text{RANK}^+}$ functionality against malicious adversaries controlling any $t < n/2$ parties.*

Proof: We just show correctness. We start with the output-ordered as in GroupBy.common, and returns it into grouped-order ($\vec{d}$). This is done by applying the inverse of π_{GNtoOUT} and so we move from $\vec{d}$ to $\vec{d}$. That is, next to each element in $\vec{k_{\text{G}}}$ that actually appears in the output order (recall that those are the last elements within each group), in $\vec{d}$ we find the number of elements in that group, and 0 in the other locations. The vector $\vec{s}$ is then the prefix-sum of $\vec{d}$. Therefore, in $\vec{s}$:

- Next to each element with key k_i that is not the last in its group, we find the total number of elements with keys smaller than k_i.
- Next to each element with key k_i that is the last one in its group, we find the total number of elements with keys smaller than or equal to k_i.

As such, s_{i-1} always contains the total number of elements with keys smaller than k_i. By taking $(i-1)$, i.e., the total number of elements until the k_i, minus s_{i-1}, i.e., the total number of elements with keys *strictly smaller* than k_i, we get the rank of k_i in its group. □

RANK$^-$. We describe the protocol in Protocol 4.14.

Protocol 4.14 $((\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{k_{\text{G}}}\rrbracket, \llbracket\vec{d}\rrbracket) \leftarrow \text{RANK}^-(\llbracket\vec{k}\rrbracket, \llbracket\vec{v}\rrbracket))$:

Input: Keys $\llbracket\vec{k}\rrbracket$ and values $\llbracket\vec{v}\rrbracket$
Output: Grouped keys $\llbracket\vec{k_{\text{G}}}\rrbracket$ and its descending rank $\llbracket\vec{d}\rrbracket$ (starting from 0).

(1) $(\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{c}\rrbracket) \leftarrow$ GroupBy.count$(\llbracket\vec{k}\rrbracket, \llbracket\vec{v}\rrbracket)$
(2) Store $\llbracket\vec{k_{\text{G}}}\rrbracket$ and $\llbracket\pi_{\vec{\text{GNtoOUT}}}\rrbracket$, which is outputs of GroupBy.common in the process of GroupBy.count
(3) For every $i \in [m-1]$: $\llbracket x_i\rrbracket = \llbracket c_{i+1}\rrbracket$, and set $x_m = \llbracket 0\rrbracket$.
(4) The parties send $(\texttt{ApplyInv}, \llbracket\pi_{\vec{\text{GNtoOUT}}}\rrbracket, \llbracket\vec{x}\rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket\vec{y}\rrbracket$.
(5) Set $\llbracket s_m\rrbracket = 0$. For every $i = m-1, \ldots, 1$ (in descending order):
  (a) $\llbracket s_i\rrbracket := \llbracket y_i\rrbracket + \llbracket s_{i+1}\rrbracket$
(6) For every $i \in [m]$: $\llbracket d_i\rrbracket = m - i - \llbracket s_i\rrbracket$
(7) Output $(\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{k_{\text{G}}}\rrbracket, \llbracket\vec{d}\rrbracket)$

An example of an execution of Protocol 4.14 is given below.

$\vec{k}$	$\vec{v}$	$\vec{k_{\text{OUT}}}$	$\vec{c}$	$\vec{x}$	$\vec{k_{\text{G}}}$	$\vec{y}$	$\vec{s}$	$\vec{d}$
2	2	1	3	2	1	0	3	2
1	3	2	2	1	1	0	3	1
1	15	3	1	0	1	2	3	0
2	4	null	0	0	2	0	1	1
3	1	null	0	0	2	1	1	0
1	10	null	0	0	3	0	0	0
Input O.		Output O.			Group O.			

Claim 4.15. *Protocol 4.14 securely computes the $\mathcal{F}_{\text{RANK}^-}$ functionality against malicious adversaries controlling any $t < n/2$ parties.*

Proof: We show correctness. We start with the output of GroupBy.count. In $\vec{x}$, each element (that appears in the output order) contains the number of elements with keys strictly greater than its key. We then move from the output order to the group order, and this is the vector $\vec{y}$. On the vector $\vec{y}$ we compute "suffix-sum" into the vector $\vec{s}$. As a result, in each s_i, each element contains the number of elements with keys greater than its key (and not including its key).

As such, s_i always contains the total number of elements with keys strictly greater than k_i. By taking $(m-i)$, i.e., the total number of elements greater (or equal) k_i, minus s_i, i.e., the total number of elements with keys strictly greater than k_i, we get the descending order rank of k_i in its group. □

FindZeroOrOne. Yet one more helping procedure for implementing GroupBy.median is FindZeroOrOne. This procedure receives set of keys $\vec{k_{\text{G}}}$ and values $\vec{v_{\text{G}}}$ in a grouped order, and a set of values $\vec{x}$ with the following guarantee: Within each group, only one element has value x_i which is 0 or 1.

To implement this procedure, we define the predicate EqualZeroOrOne$(\llbracket a\rrbracket)$ on some shared value a, which returns $\llbracket 1\rrbracket$ if $a = 0$ or 1, and it returns $\llbracket 0\rrbracket$ otherwise. We discuss how to implement this procedure below. By the input assumption on $\vec{x}$, if we apply EqualZeroOrOne on each element x_i, exactly one element in each group would result with 1. We then return, in an output order, that value of the element with the indicator 1. We implement FindZeroOrOne in Protocol 4.16.

Protocol 4.16 $(\llbracket\vec{w}\rrbracket \leftarrow \text{FindZeroOrOne}(\llbracket\vec{k_{\text{G}}}\rrbracket, \llbracket\vec{v_{\text{G}}}\rrbracket, \llbracket\vec{x}\rrbracket))$:

Input: Grouped keys $\llbracket\vec{k_{\text{G}}}\rrbracket$, $\llbracket\vec{v_{\text{G}}}\rrbracket$ and values $\llbracket\vec{x}\rrbracket$ with the guarantee that exactly one element in each group has $x_i = 0$ or $x_i = 1$.
Output: The values that correspond to $x_i = 0$ or $x_i = 1$ in a grouped order.

(1) For every $i \in [m]$, set $\llbracket\alpha_i\rrbracket =$ EqualZeroOrOne$(\llbracket x_i\rrbracket)$.
(2) For every $i \in [m]$, the parties send $(\alpha_i, v_{\text{G}}[i])$ to $\mathcal{F}_{\text{mult}}$ and receive $\vec{z}$.
(3) The parties send $(\text{GenPerm}, 1 - \llbracket\alpha_i\rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive back $\llbracket\pi\rrbracket$.
(4) The parties send $(\texttt{ApplyPerm}, \llbracket\pi\rrbracket, \llbracket z\rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket\vec{w}\rrbracket$.
(5) Output $\llbracket\vec{w}\rrbracket$.

EqualZeroOrOne$(\llbracket a\rrbracket)$: A simple way to implement this procedure it to truncate the LSB from a, and then run equall$(\llbracket a'\rrbracket, 0)$

(where a' is the truncated a). Alternatively, one can check equality to 0 and 1 separately, and then run a circuit that computes the OR operation.

The following claim follows easily by inspection and by the correctness of the underlying primitives ($\mathcal{F}_{\text{sort}}$, EqualZeroOrOne, etc.):

Claim 4.17. *Protocol 4.16 securely computes $\mathcal{F}_{\text{FindZeroOrOne}}$ against malicious adversaries controlling any $t < n/2$ parties.*

4.7 Generalizing group-by-median into quartiles and percentiles

Quartiles and percentiles are generalizations of the median, and are used to analyze the shape of the distribution, especially the dispersion, in a way that is resistant to outliers.

We generalize the group-by-median protocol to obtain percentiles. More precisely, for each group of c elements the protocol computes the $(\frac{t}{t+u}(c-1)+1)$-th element.[5] This is the median if $t = u = 1$, quartiles if $t = i$ and $u = 4 - i$ for $1 \le i \le 3$, and percentiles if $t = j$ and $u = 100 - j$ for $0 \le j \le 100$.

If the number of elements in the group minus one, i.e., the number of intervals between elements, $c - 1$, is not divisible by $t + u$, then let $\tau = \frac{t}{t+u}(c-1)+1$ and $\delta_\tau = \lceil\tau\rceil - \tau$, and let the τ-th element in $v_1, \ldots, v_c$ be the interpolated value, $v^* = \delta_\tau v_{\lceil\tau\rceil-1} + (1-\delta_\tau)v_{\lceil\tau\rceil}$. Similar to GroupBy.median, our GroupBy.percentile protocol outputs $(t+u)v^*$ instead of v^* to avoid Real-number operations, i.e., dividing by $t+u$. We can easily extend our protocol to compute v^* directly by using a division protocol [6].

We show the generalized protocol GroupBy.percentile in Protocol 4.10.

Protocol 4.18 $((\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{z}\rrbracket) \leftarrow \text{GroupBy.percentile}(t, u, \llbracket\vec{k}\rrbracket, \llbracket\vec{v}\rrbracket))$**:**

Input: Parameters t and u, and keys $\llbracket\vec{k}\rrbracket$ and values $\llbracket\vec{v}\rrbracket$
Output: Grouped keys $\llbracket\vec{k_{\text{G}}}\rrbracket$ and its $\frac{t}{t+u}$-th element $\llbracket\vec{z}\rrbracket$

(1) $(\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{k_{\text{G}}}\rrbracket, \llbracket\vec{a}\rrbracket) \leftarrow \text{rank}^+(\llbracket\vec{k}\rrbracket, \llbracket\vec{v}\rrbracket)$.
(2) $(\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{k_{\text{G}}}\rrbracket, \llbracket\vec{d}\rrbracket) \leftarrow \text{rank}^-(\llbracket\vec{k}\rrbracket, \llbracket\vec{v}\rrbracket)$.
(3) For every $i \in [m]$:
  (a) Invoke $\llbracket f_i\rrbracket \leftarrow \text{isInRange}(u\cdot\llbracket a_i\rrbracket, t\cdot\llbracket d_i\rrbracket, t\cdot\llbracket d_i\rrbracket + t + u)$.
  (b) The parties send $(u\llbracket a_i\rrbracket - t\llbracket d_i\rrbracket, \llbracket v_{i-1}\rrbracket)$ to $\mathcal{F}_{\text{mult}}$ and receive $\llbracket x_i\rrbracket$
  (c) The parties send $(t+u-u\llbracket a_i\rrbracket + t\llbracket d_i\rrbracket, \llbracket v_i\rrbracket)$ to $\mathcal{F}_{\text{mult}}$ and receive $\llbracket y_i\rrbracket$
  (d) The parties send $(\llbracket x_i\rrbracket + \llbracket y_i\rrbracket, \llbracket f_i\rrbracket)$ to $\mathcal{F}_{\text{mult}}$ and receive $\llbracket z_i\rrbracket$
(4) The parties send $(\texttt{GenPerm}, 1 - \llbracket\vec{f}\rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket\vec{\rho}\rrbracket$.
(5) The parties send $(\texttt{ApplyPerm}, \llbracket\vec{\rho}\rrbracket, \llbracket\vec{z}\rrbracket)$ to $\mathcal{F}_{\text{sort}}$ (Functionality 2.1) and receive $\llbracket\vec{z_{\text{OUT}}}\rrbracket$
(6) Output $(\llbracket\vec{k_{\text{OUT}}}\rrbracket, \llbracket\vec{z_{\text{OUT}}}\rrbracket)$

The main idea is that the $(\frac{t}{t+u}(c-1)+1)$-th element divides the intervals between the grouped c elements in a ratio of t to u. Therefore, if we multiply the outputs of rank^- and rank^+ by t and u, respectively, then the point at which the large/small relationship between the two values is swapped is the $(\frac{t}{t+u}(c-1)+1)$-th element. Using this observation, the protocol first computes the flag $\vec{f}$ in which 1 appears only at the $\lceil\frac{t}{t+u}(c-1)+1\rceil$-th element in each group, where c is the count of each group. We explain below in more detail how to compute this flag.

Let $\vec{a}$ and $\vec{d}$ be the ascending and descending orders computed by rank^+ and rank^-. Let the i-th record among all records for $1 \le i \le m$ be the j-th record in a group whose count is c (and therefore $1 \le j \le c$). Let $a_i = j-1$, $d_i = c-j$. Here, we want to set $f_i = 1$ iff $j = \lceil\frac{t}{t+u}(c-1)+1\rceil$ holds. We observe that

$$\begin{aligned} & j = \lceil\frac{t}{t+u}(c-1)+1\rceil \\ \Leftrightarrow\ & j-1 < \frac{t}{t+u}(c-1)+1 \ \wedge\ \frac{t}{t+u}(c-1)+1 \le j \\ \Leftrightarrow\ & u(j-1) < t(c-j)+(t+u) \ \wedge\ t(c-j) \le u(j-1) \\ \Leftrightarrow\ & ua_i < td_i + (t+u) \ \wedge\ td_i \le ua_i, \end{aligned}$$

which can be computed from $u\llbracket\vec{a}\rrbracket$ and $t\llbracket\vec{d}\rrbracket$.

After obtaining the flag $\vec{f}$ representing the $\lceil\frac{t}{t+u}(c-1)+1\rceil$-th element in each group, the remaining task is computing the output of $(t+u)$ times the interpolated value, $v^* = \delta_\tau v_{\lceil\tau\rceil-1} + (1-\delta_\tau)v_{\lceil\tau\rceil}$, where $\tau = \frac{t}{t+u}(c-1)+1$ and $\delta_\tau = \lceil\tau\rceil - \tau$. Recall that if $f_i = 1$ and $\lceil\tau\rceil = j$, $a_i = j-1$, and $d_i = c-j$ hold, then

$$\delta_\tau = j - \tau = j - \left(\frac{t}{t+u}(c-1)+1\right) = \frac{tj+uj-tc-u}{t+u} = \frac{ua_i - td_i}{t+u}.$$

Therefore, the output of GroupBy.percentile (which is $(t+u)v^*$ to avoid real-number operation) is multiplying

$$(t+u)v^* = (ua_i - td_i)v_{i-1} + (t+u-ua_i+td_i)v_i$$

with f_i to filter rows with $f_i = 0$.

Protocol 4.19 $(\llbracket f\rrbracket \leftarrow \text{isInRange}(\llbracket a\rrbracket, \llbracket r_0\rrbracket, \llbracket r_1\rrbracket))$**:**

Input: Shares of a, r_0 and r_1.
Output: A shared bit f, where $f = 1$ if $r_0 \le a < r_1$ and 0 otherwise.

(1) Invoke $\llbracket e_0\rrbracket \leftarrow \text{lessthan}(\llbracket a\rrbracket, \llbracket r_0\rrbracket)$.
(2) Invoke $\llbracket e_1\rrbracket \leftarrow \text{lessthan}(\llbracket a\rrbracket, \llbracket r_1\rrbracket)$.
(3) The parties send $(1 - \llbracket e_0\rrbracket, \llbracket e_1\rrbracket)$ to $\mathcal{F}_{\text{mult}}$ and receive $\llbracket f\rrbracket$.
(4) Output $(\llbracket f\rrbracket)$

An example of an execution when $t = 1$ and $u = 3$ (computing first quartile) is shown in Table 5.

We next show how to implement the isInRange procedure.

The is done by first comparing the input a to the lower bound and upper bound inputs, using the lessthan procedure. This produces two shared bits $\llbracket e_0\rrbracket$ and $\llbracket e_1\rrbracket$. Note that $e_0 = 1$ if $a < r_0$ and 0 otherwise, and $e_1 = 1$ if $a < r_1$ and 0 otherwise. Hence, multiplying $1 - e_0$ and e_1 yield 1 if and only if $r_0 \le a < r_1$ as required.

4.8 Removing Null rows

As in the case of the join protocols, we have the option to remove Null rows. For Group-by protocols, removing Null rows can be conducted as in the join protocols since $\llbracket\vec{e}\rrbracket$ in group-by-common can be regarded as the valid bit.

[5] This definition is the one used in the python library NumPy and in the PERCENTILE.INC function in Microsoft Excel.

$\vec{k}$	$\vec{v}$	$\vec{k_G}$	$\vec{v_G}$	$\vec{a}$	$\vec{d}$	$\vec{f}$	$u\vec{a} - t\vec{d}$	$\vec{x}$	$\vec{y}$	$\vec{z}$	$\vec{k_{OUT}}$	$\vec{z_{OUT}}$
1	2	1	2	0	1	0	-1	0	10	0	1	9
3	4	1	3	1	0	1	3	6	3	9	2	4
1	3	2	1	0	0	1	0	0	4	4	3	17
3	5	3	4	0	1	0	-1	-1	20	0	null	0
2	1	3	5	1	0	1	3	12	5	17	null	0
Input O.							Grouped O.				Output O.	

Table 5: Process example of GroupBy.percentile when $t = 1$ and $u = 3$ (first quartile). For the key 1 there are $c = 2$ elements, $\tau = \frac{t}{t+u}(c-1) + 1 = 1 + 1/4$, and $\delta_\tau = \lceil \tau \rceil - \tau = 3/4$. The protocol computes the τth element in the group $\{2, 3\}$, which is $v^* = \delta_\tau \cdot v_{\lceil \tau \rceil - 1} + (1 - \delta_\tau) v_{\lceil \tau \rceil} = \frac{3}{4} v_1 + \frac{1}{4} v_2 = 2\frac{1}{4}$. The protocol outputs $(t+u)v^* = 4 \cdot 2\frac{1}{4} = 9$ instead of v^* to avoid real-number operations.

	$\mathcal{F}_{\text{mult}}$	$\mathcal{F}_{\text{sort}}$		Equality Check
		Gen(m)	apply(m)	
GroupBy.common	m	3	3	m
GroupBy.count	-	-	1	-
GroupBy.sum	m	-	1	-
GroupBy.max	m	-	1	-
GroupBy.min	m	1	1	-
GroupBy.median	$2m$	2	5	$4m$

Table 6: Costs of our group-by protocols, measured by the number of calls to the underlying primitives: multiplication, generating a permutation, applying a permutation and checking equality between two shared values. We note that the cost of GroupBy.common is not included in the cost of each of the other operations, since it can be called once for all operations.

Operation	Protocol	Round Complexity
Join-uu	[23]	$r_{\text{prf}} + r_{\text{eq}} + c_1$
	Ours	$r_{\text{gp}} + c_{\text{ap}} + c_{\text{ai}} + r_{\text{eq}} + c_2$
Join-un	[8]	$r_{\text{gp}} + c_{\text{ap}} + c_{\text{ai}} + 2\log(n+m) + c_3$
	Ours	$r_{\text{gp}} + c_{\text{ap}} + c_{\text{ai}} + c_4$

Table 7: Comparison of round complexity with previous protocols in the semi-honest security model. m, n refer to the sizes of the left and right tables, respectively. r_{prf} refers to the round complexity of computing pseudo-random function (LowMC in [23]) on shared inputs. r_{eq} refers to the round complexity of equality check, which depends on the algorithm. $r_{\text{gp}}, c_{\text{ap}}, c_{\text{ai}}$ refer to the round complexity of GenPerm, ApplyPerm, ApplyInv, respectively. Note that $r_{\text{gp}} = O(\ell)$ and independent of m, n, where ℓ is the bit length of the sorting keys. c_1, c_2, c_3, c_4 are additional small constants. We can assume that $c_{\text{xx}} \leq 5$.

4.9 Cost analysis

Table 6 summarizes the costs of our group-by protocols. For each protocol, we present the number of calls to each of the underlying sub-protocols that we use. Since GroupBy.common is used in each of the group-by operations, we computed its cost separately. If one computes many statistics over a data-set, it suffices to run GroupBy.common once and feed its output to all other operations.

5 EXPERIMENTAL RESULTS

This section describes the experimental results of our implementation of the join and group-by protocols, for the setting of *three* servers and *one* single corrupted party.

Setting. For the building block of our protocols, we used the recent three-party stable sorting protocol of Asharov et al. [5], which has communication that grows linearly with the size of the data sets, and the number of rounds is logarithmic in the key length. For multiplication, we used the protocol of Araki et al. [28], where each party sends only one field element per multiplication. Our secret sharing was defined over the field $\mathbb{F}_{2^{61}-1}$, which supports faster multiplication as $2^{61} - 1$ is a Mersenne prime. To achieve malicious security, we used the technique used in [1, 12, 19], where the parties hold a MAC over each secret. This requires them to invoke the multiplication protocol twice for each multiplication operation in the protocol, to maintain the MAC. We note that this method achieves statistical security. In our implementation, the statistical security is roughly 2^{-30}. The reason for this is that in the sorting protocol of [5], the elements are decomposed to their bit representation. This requires us to maintain a MAC over each secret shared bit. We used the field $\mathbb{F}_{2^{32}}$ in the MAC for a small field. By the analysis of [12], this implies success cheating probability of $2/(2^{31} - 1) \approx 2^{-30}$.

We have implemented our protocols in C++11. Our experiments were performed in two experimental environments, denoted by *HI* and *LO*. In the experiments performed with HI, the parameters were set up to achieve the best possible performance. In contrast, the LO environment is used mainly to compare our results with previous works. HI consists of three servers connected by a 10 Gbps LAN with an RTT of 0.136 ms. Each server has two Intel Xeon Gold 6144k (3.50GHz 8 cores/16 threads) CPUs and 765GB of memory. LO consists of three computers connected by a 1 Gbps LAN with an RTT of 0.362 ms. Each computer has an Intel Core i7-6700 (3.40GHz 4 cores/16 threads) CPU and 32GB memory, which is roughly the same setting used in [8, 23].

Join protocols.

We first measured the running times of our join protocols. We consider table sizes of $m \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$. In our measurements, the number of rows in both tables was m, and each table had a column of 32-bit keys and a column of 50-bit values.

We present in Table 7 the round complexity and in Table 8 the running times of our join protocols and previous works. Each running time for our protocols is an average of five measurements. The running times and total communication for [23] and [8] are taken from the corresponding papers for comparison.

The experimental results on HI show that our implementation achieves very fast joins on three-servers. For example, joining tables

Operation	Security	Protocol	Time (sec.) $m=2^8$	$m=2^{12}$	$m=2^{16}$	$m=2^{20}$	Total Communication (MB) $m=2^8$	$m=2^{12}$	$m=2^{16}$	$m=2^{20}$
Join-uu	SH	[23]	0.02	0.03	0.3	9.1	0.3	4.9	78.1	1249.4
	SH	Our (HI)	0.030	0.037	0.185	3.850	0.48	12.19	193.47	3094.36
	SH	Our (LO)	0.027	0.069	0.908	15.311				
	Mal.	Our (HI)	0.220	0.216	0.638	10.642	3.17	59.29	941.37	15054.26
	Mal.	Our (LO)	0.127	0.298	3.827	61.195				
Join-un	SH	[8]	0.09	0.21	1.3	21.6	1.5	22.8	364	5560
	SH	Our (HI)	0.021	0.039	0.240	5.463	0.64	17.81	283.73	4538.23
	SH	Our (LO)	0.023	0.094	1.294	21.764				
	Mal.	Our (HI)	0.160	0.182	0.794	17.978	3.65	75.57	1202.98	19241.31
	Mal.	Our (LO)	0.099	0.344	4.746	77.139				

Table 8: The running times in seconds and communication overhead in MB for inner join. SH and Mal. refer to Semi-honest and Malicious. In the experiments, both tables have m rows and two columns (one for the key and the other for associated value).

of 2^{20} records takes 3.850 seconds and 5.463 seconds with semi-honest security when the join operation is join-uu and join-un, respectively. It takes only 10.642 seconds for join-uu and 17.978 seconds for join-un with malicious security. Overall, the running times of our protocols with malicious security are 2.76–4.00 times larger than those with semi-honest security for $m = 2^{20}$.

It is important to note that the LowMC cipher [2] that was used in the experiment in [23] has later been cryptanalyzed [22]. The overhead of our protocol does not depend on the implementation of a specific cipher, and therefore there is no need to use high performance ciphers that might have low security.

Our semi-honest protocols have comparable performance to those of existing works. Although a fair comparison is difficult due to the different measurement environments, for semi-honest security, both [23] and [8] results are between our LO and HI results.

We are the first to implement join protocols secure against a malicious adversary. Finally, for join-un, we have a left-outer join which is non-trivial to obtain from the join protocol in [8].

We also measured the communication overhead of our join protocols in the same setup as the running time measurements. Since the communication overhead does not depend on the servers, we measured it only on HI. We present in Table 8 the communication overhead of our join protocols and previous works. The communication overhead for [23] and [8] are taken from the corresponding papers for comparison.

For join-uu, our semi-honest protocol required about 2.5 times as much communication as [23]. On the other hand, for join-un, our semi-honest protocol required only about 0.82 times as much communication as [8]. The communication overhead of our malicious security protocol was about 4.9 and 4.3 times higher than that of our semi-honest protocol for join-uu and join-un, respectively.

Group-by protocols. We also measured the running times of our implementations of group-by protocols. Our implementation of GroupBy.count and GroupBy.sum, used a modified and more efficient GroupBy.common protocol that does not use $[[\vec{v}]]$ as input to the sorting operation (but rather only the key column); see implementation note in Section 4.1. We again consider table sizes of $m \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$. In our measurements the table has m rows, a column of 32-bit keys and a column of 50-bit values.

Operation	Security	Time (sec.) $m=2^8$	$m=2^{12}$	$m=2^{16}$	$m=2^{20}$
GroupBy.common	SH	0.054	0.068	0.251	3.740
	Mal.	0.440	0.449	0.794	9.979
GroupBy.common (without sort by $[[\vec{v}]]$)	SH	0.023	0.026	0.098	1.550
	Mal.	0.188	0.197	0.356	4.744
GroupBy.max*	SH	0.001	0.001	0.005	0.094
	Mal.	0.002	0.002	0.007	0.125
GroupBy.min*	SH	0.001	0.002	0.007	0.115
	Mal.	0.004	0.004	0.015	0.189
GroupBy.median*	SH	0.006	0.007	0.020	0.302
	Mal.	0.057	0.084	0.188	2.575
GroupBy.count**	SH	0.001	0.001	0.004	0.069
	Mal.	0.001	0.001	0.005	0.096
GroupBy.sum**	SH	0.001	0.001	0.004	0.070
	Mal.	0.001	0.002	0.008	0.150

Table 9: The running times in seconds for group-by operations measured in the HI setting. * and ** denote the running times of GroupBy.common and its modified version (excluding $[[\vec{v}]]$ for sorting), respectively. In the Security column, SH and Mal. refer to Semi-honest and Malicious.

Operation	Security	Time (sec.) $m=2^8$	$m=2^{12}$	$m=2^{16}$	$m=2^{20}$
GroupBy.common	SH	0.043	0.061	1.046	17.245
	Mal.	0.257	0.345	4.216	66.422
GroupBy.common (without sort by $[[\vec{v}]]$)	SH	0.020	0.028	0.438	7.196
	Mal.	0.115	0.157	1.889	29.877
GroupBy.max*	SH	0.000	0.001	0.021	0.345
	Mal.	0.002	0.003	0.044	0.678
GroupBy.min*	SH	0.001	0.002	0.032	0.548
	Mal.	0.003	0.005	0.074	1.112
GroupBy.median*	SH	0.004	0.009	0.093	1.635
	Mal.	0.041	0.077	0.810	14.413
GroupBy.count**	SH	0.001	0.001	0.014	0.220
	Mal.	0.001	0.003	0.032	0.522
GroupBy.sum**	SH	0.001	0.001	0.013	0.206
	Mal.	0.001	0.003	0.046	0.727

Table 10: The running times in seconds for group-by operations measured in the LO setting. The abbreviations and symbols are the same as in Table 9.

The results for HI and LO are presented in Tables 9 and 10, respectively. In the tables, the running time of GroupBy.common is excluded from the run time of the other group-by protocols, since group-by protocols often computes multiple statistics at the same time, which requires only a single invocation of GroupBy.common.

The results in Table 9 show extremely fast group-by operations. Even for a table with 2^{20} records, and even for the most time-consuming group-by-median operation, the output is computed in 4.042 (= 3.740 + 0.302) seconds with semi-honest security and in 12.554 (= 9.979 + 2.575) seconds with malicious security. Our protocols are advantageous when computing several types of statistics simultaneously. For example, computing all group-by statistics of max, min, median, count and sum, takes 4.390 seconds for the semi-honest security and 13.114 seconds for the malicious security. The running times in LO are 2.94–8.06 times larger than those in HI when $m = 2^{20}$. The difference in time between malicious and semi-honest in GroupBy.median is comparably large. This is expected to be due to the fact that our implementation of GroupBy.median contains many bit operations which incur a large overhead when using the compiler of [12].

ACKNOWLEDGEMENTS

Asharov is sponsored by the Israel Science Foundation (grant No. 2439/20), by JPM Faculty Research Award, and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 891234.

REFERENCES

[1] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. 2021. An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings. In *ACNS 2021 (Lecture Notes in Computer Science, Vol. 12727)*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer, 122–152. https://doi.org/10.1007/978-3-030-78375-4_6

[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9056. Springer, 430–454. https://doi.org/10.1007/978-3-662-46800-5_17

[3] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptogr.* 5, 4 (2021), 27. https://doi.org/10.3390/cryptography5040027

[4] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 610–629. https://doi.org/10.1145/3460120.3484560

[5] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 125–138. https://doi.org/10.1145/3548606.3560691

[6] Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Takahiro Matsuda, Ibuki Mishina, Hiraku Morita, and Jacob C. N. Schuldt. 2022. Adam in Private: Secure and Fast Training of Deep Neural Networks with Adaptive Moment Estimation. *Proc. Priv. Enhancing Technol.* 2022, 4 (2022), 746–767.

[7] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, Tadanori Teruya, and Kazunari Tozawa. 2022. Secure Parallel Computation on Privately Partitioned Data and Applications. In *CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 151–164. https://doi.org/10.1145/3548606.3560695

[8] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[9] Marina Blanton and Everaldo Aguiar. 2012. Private and oblivious set and multiset operations. In *ASIACCS '12*. 40–41.

[10] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Trans. Computers* 38, 11 (1989), 1526–1538.

[11] Guy E. Blelloch. 1990. Prefix sums and their applications. https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

[12] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO 2018 (Lecture Notes in Computer Science, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 34–64. https://doi.org/10.1007/978-3-319-96878-0_2

[13] R. Cramer, I. Damgård, and Y. Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC (LNCS, Vol. 3378)*. Springer, 342–362.

[14] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM, 79–88.

[15] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient Private Matching and Set Intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, 1–19.

[16] O. Goldreich. 2004. *Foundations of Cryptography*.

[17] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. 2023. Efficient decision tree training with new data structure for secure multi-party computation. *Proc. Priv. Enhancing Technol.* 2023, 1 (2023), 343–364. https://doi.org/10.56553/popets-2023-0021

[18] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.

[19] Ryo Kikuchi, Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ai Ishida, Takahiro Matsuda, Yusuke Sakai, and Jacob C. N. Schuldt. 2019. Field Extension in Secret-Shared Form and Its Applications to Efficient Secure Computation. In *ACISP 2019 (Lecture Notes in Computer Science, Vol. 11547)*, Julian Jang-Jaccard and Fuchun Guo (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-030-21548-4_19

[20] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. 2018. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *ACISP 2018*. 64–82.

[21] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *ACNS 2013 (Lecture Notes in Computer Science, Vol. 7954)*, Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer, 84–101. https://doi.org/10.1007/978-3-642-38980-1_6

[22] Fukang Liu, Takanori Isobe, and Willi Meier. 2021. Cryptanalysis of Full LowMC and LowMC-M with Algebraic Techniques. In *CRYPTO 2021 (Lecture Notes in Computer Science, Vol. 12827)*, Tal Malkin and Chris Peikert (Eds.). Springer, 368–401. https://doi.org/10.1007/978-3-030-84252-9_13

[23] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[24] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *SP 2015*. IEEE Computer Society, 377–394. https://doi.org/10.1109/SP.2015.30

[25] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 515–530.

[26] A. Shamir. 1980. On the Power of Commutativity in Cryptography. In *ICALP*.

[27] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*. IEEE Computer Society, 44–55.

[28] T.Araki, J.Furukawa, Y.Lindell, A.Nof, and K.Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*. ACM, 805–817.