Secret-Shared Joins with Multiplicity from Aggregation Trees

Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella,
Srinivasan Raghuramam and Peter Rindal

Abstract

We present novel protocols to compute SQL-like join operations on secret shared database tables with *non-unique* join keys. Previous approaches to the problem had the restriction that the join keys of both the input tables must be unique or had quadratic overhead. Our work lifts this restriction, allowing one or both of the secret shared input tables to have an unknown and unbounded number of repeating join keys while achieving efficient $O(n \log n)$ asymptotic communication/computation and $O(\log n)$ rounds of interaction, *independent* of the multiplicity of the keys.

We present two join protocols, $\Pi_{\mathsf{Join\text{-}OM}}$ and $\Pi_{\mathsf{Join\text{-}MM}}$. The first, $\Pi_{\mathsf{Join\text{-}OM}}$ is optimized for the case where one table has a unique primary key while the second, $\Pi_{\mathsf{Join\text{-}MM}}$ is for the more general setting where both tables contain duplicate keys. Both protocols require $O(n \log n)$ time and $O(\log n)$ rounds to join two tables of size n. Our framework for computing joins requires an efficient sorting protocol and generic secure computation for circuits. We concretely instantiate our protocols in the honest majority three-party setting.

Our join protocols are built around an efficient method to compute structured aggregations over a secret shared input vector $\mathcal{V} \in \mathbb{D}^n$. If the parties have another secret-shared vector of control bits $\mathcal{B} \in \{0,1\}^n$ to partition $\mathcal{V}$ into sub-vectors (that semantically relates to the join operations). A structured aggregation computes a secret shared vector $\mathcal{V}' \in \mathbb{D}^n$ where every sub-vector $(\mathcal{V}_b, ..., \mathcal{V}_e)$ (defined by the control bits) is aggregated as $\mathcal{V}'_i = \mathcal{V}_b \star ... \star \mathcal{V}_i$ for $i \in \{b, ..., e\}$ according to some user-defined operator $\star$. Critically, the b, e indices that partition the vector are secret. It's trivial to compute aggregations by sequentially processing the input vector and control bits. This would require $O(n)$ rounds and would be very slow due to network latency. We introduce Aggregation Trees as a general technique to compute aggregations in $O(\log n)$ rounds. For our purpose of computing joins, we instantiate $\star \in \{\mathsf{copy\ previous\ value}, \mathsf{add}\}$, but we believe that this technique is quite powerful and can find applications in other useful settings.

1 Introduction

We advance the state of art for performing *c*omposable SQL-style join operations on secret shared input databases. Given two secret shared database tables, our protocols constructs a secret shared table containing a join of the two tables. Our protocol lifts all restrictions on the join columns to efficiently compute the join on key columns which have an *unbounded and unknown* number of duplicates. Previous protocols [MRR20, LTW13a] for this problem with efficient asymptotic communication have required the join keys to be unique. Our protocols require $O(\log n)$ rounds and has computation and communication that is $O(n \log n)$, where n is the size of the input/output tables. Our protocol can handle/perform the full suite of join operations including inner, union, left and full joins(see Appendix A for definitions). Our protocol is highly robust and can be instantiated in any semi-honest setting with an efficient secret-shared sorting and generic secure computation

protocol. We concretely implement our protocol in the semi-honest, three party setting with an honest-majority.

In recent years, we have seen a lot of exciting work towards privately computing [PSZ14a, PSSZ15a, PSZ16, KKRT16, PSWW18, CLR17, CHLR18, IKN+17a, RA18, KLS+17, OOS17, KMP+17, MRR20, RS21, RR22] set intersection, union, inner join, secret-shares of set intersection and related functionalities, which have shown great promise for practical deployment. A vast majority of these works target the special-case of *private set intersection* (PSI), which is analogous to revealing the entire result of an inner join. However, real-world applications require more general and robust properties. First, we would like to compute any type of join/functionality *without revealing the result* (to enable performing further joins, filtering, or computing some aggregate information). This will also require the inputs to be secret-shared since the input might be the output of a previous MPC computation. Finally, most existing works assume that inputs (join-keys) are unique which is not the case with real-world data. Designing efficient protocols for this general task of *composable SQL* with secret-shared inputs, outputs and *non-unique* join keys is significantly harder, and techniques from traditional setting of PSI do not automatically translate.

The first important property of our protocols is that the inputs/outputs are secret shared and therefore can be *composed* together, where the output of a join can be the input to another. Allowing this composability greatly increases the ability to perform highly complex queries and enables additional computation to be performed after the join. For example, most privacy preserving machine learning publications [MR18a, Fac20, DPDea20, WGC19, RSC+19] assume that the data being trained on has already been joined together. Without our line of work, it is unclear how real-world data can be filtered and joined for these machine learning tasks.

Prior privacy preserving join techniques with good practical performance assume unique join keys [MRR20] or were too computationally expensive [LTW13a] to be used at scale in practice. Moreover, many practical application require the ability to efficiently handle duplicates or multiplicity over the join keys. Sometimes, inputs inherently have one-to-many or many-to-many relationships. For example, one party has a set of users with associated data, while the other party has input data with respect to a set of groups. In this case we want to join the group data with the user data. This is precisely a one-to-many join. Now, for whatever reason, if we need to construct a table for all pairs of users within each group, we can express this as a many-to-many join between the two tables. Until now, being able to efficiently run join queries over such relations on large inputs has been all but impossible in the secret shared setting.

Queries of this type often come up in practice. One recent example has been the push to develop privacy preserving ads [Wil19, Met22, IKN+17b]. This application aims to, in *aggregate*, attribute someone buying an advertised product (possibly offline) to them clicking on an ad on an online platform without revealing information about individual users. This process involves linking data between several entities and, when done at scale, requires both one-to-many and many-to-many joins between distrusting parties, which are ideally performed using MPC. Another application is financial fraud detection. Companies are legally mandated to implement methods for identifying and preventing fraud. However, the necessary information for determining this is often spread across several institution which necessitates some form of data sharing [SvHA+19]. Moreover, the types of data analysis involved often require a series of joins on non-unique attributes. For example, joining merchant data with issuer data for each transaction and then forwarding the join into a machine learning classifier [CTB+18]. Even this simple use case would involve both a one-to-many and many-to-many join.

Computing joins efficiently with one-to-many or many-to-many relations has many challenges. The first is identifying a mechanism to match up the records. Traditionally, this has been done using a hash table where the row with join key k is (obliviously) mapped to the hash table position $h(k)$. Then it is a simpler task of comparing the items mapped to the same position. However, it is unclear how such a strategy would work in the case of duplicate keys. In the worst case, all n rows of a table might have the same key k, all of them would be mapped to the same position leaking the fact that all items have the same key. Therefore, an approach that does not use a hash table appears necessary.

One could construct the join by comparing all n^2 pairs of input rows. However this would be impractical for large n. Alternatively, one could imagine there where exists a circuit that more cleverly compares the tables. However, as we discuss later, many of the natural approaches to compute joins without performing n^2 work requires a circuit of depth $O(n)$, i.e. a circuit with linear multiplicative depth. Then if a non-constant round MPC protocol is used (for computational efficiency), the overall round complexity would be linear. To understand the impact of this, consider the case where sending a message has a relatively fast 10 milliseconds latency, then joining two tables with $n = 1,000,000$ rows would at least requires 2.8 hours. This is in contrast with our protocols which require $O(\log n)$ rounds, e.g. about 0.2 seconds.

In this work, our main contribution is novel and efficient techniques to resolve queries over one-to-many or many-to-many relations. We present two join protocols - $\Pi_{\mathsf{Join\text{-}OM}}$ is optimized for the setting where only one of the input tables contains duplicate join keys. In this situation the output table has a maximum possible size of n where n is the size of the input tables. $\Pi_{\mathsf{Join\text{-}OM}}$ will always produce such an output table by adding dummy rows until it is of size n. The dummy rows can be removed if leaking the size is OK.

For the case with both input table might contain duplicates we present the more general protocol, $\Pi_{\mathsf{Join\text{-}MM}}$. In this setting the maximum possible output table size is n^2, i.e. when both input tables have n rows with all the same key. In this situation we optionally allow for an upper bound $D \le n^2$ on the size of the output table to be revealed to the parties. In this case the running time of the protocol is $O((D+n)\log(D+n))$. We note that D need not be known ahead of time, instead it can be computed by the protocol as a function of the input tables.

Our Contributions

- First we introduce aggregation trees, a very versatile and powerful technique for updating values in a vector based on its neighbors. In particular, the technique allows the parties to break an input vector $\mathcal{V}$ into subvectors and then for each to effectively perform a linear pass over it to update each value based on the sum of all of the predecessors (prefix) and/or successors (suffix). Most importantly, this can all be achieved in $O(\log n)$ rounds, $O(n)$ time.

- An innovative and efficient protocol $\Pi_{\mathsf{Join\text{-}OM}}$ for joining two tables that have one-to-many relation, i.e. one table is known to have unique keys. The protocol runs in time $O(n \log n)$ where n is the number of input rows in the tables and requires $O(\log n)$ rounds. It makes use of oblivious sorting, permutations and aggregation trees. For $n = 1,000,000$, the running time is 21 seconds, just 1.7× slower than the state of art one-to-one join relation protocol of [MRR20].

- Using similar but more advanced techniques, we then present the generalized protocol $\Pi_{\mathsf{Join\text{-}MM}}$ that can join tables with a many-to-many relation, i.e. both tables can have duplicate keys.
- We introduce the Aggregate-Sort-Compute pattern, a general technique for efficient data processing. In particular, we observe that this pattern enables many computations that traditionally would have required expensive techniques such as oblivious RAM.
- Composable protocols where the input and output tables are secret shared, enabling additional secret shared computation.
- Our techniques are generic and can be applied to many settings, e.g. two party, many party, malicious, etc.
- We implement our protocol in the honest majority three party framework of [MR18a]. We observe very competitive running times compared to prior protocols while offering more functionality.

2 Our Setting and Related Work

Database Joins. We want to obliviously compute database joins which have a one-to-many or many-to-many relations. We specifically, instantiate the protocol in the three-party honest majority setting. Let's look at works that are closely related to ours.

Ion et al. presented a private set intersection sum protocol [IKN+17a] that is used by Google Adwords to correlate online advertising with offline sales in a privacy preserving manner. In particular, a sum over the sales in the inner join is revealed. Pinkas et al. [PSWW18] also presented a protocol that allows the output of the join to be in a secret shared format such that the output can further be used for additional secret shared computation. These protocols can be framed in terms of SQL queries consisting of an inner join followed by an aggregation on the resulting table, e.g. summing a column. Neither of these protocols (and almost no prior related results) support secret-shared inputs, but rather require the source tables to be held in the clear by each party.

Blanton et al. [BA11] is one of the first works to propose composable protocols (with secret-shared inputs and outputs) for a comprehensive suite of set operations in the honest majority setting. Their protocols require asymptotic $\mathcal{O}(n \log n)$ communication and computation complexity where n is the total number of items in both the input sets. The core of their approach is to combine all the inputs and obliviously sort this combined list. Next, they employ generic MPC techniques to compute set operations. This is very close to our approach, except that we extend beyond set operations and look at the challenges of computing database joins in a round efficient manner.

Laur et al. in [LTW13b] present a protocol for database joins/unions in the honest majority setting with secret-shared input databases. The protocol requires an oblivious shuffle and mechanism for oblivious PRF (pseudo-random function) evaluation of secret shared inputs with secret-shared keys. The main drawback of this approach is that it leaks the size of the intersection and the result of the where clause of the join for each row to every party. As a result, this protocol is not composable, i.e., the output cannot be used as the input the next join without significant leakage. In particular, if there are *non-unique join or repeating* keys, the protocol leaks the size of the output join table and the frequency of occurrence of each key while hiding the true identity of the key. In comparison, our work only leaks the size of the output join table. Concretely, [MRR20] estimates

that their protocol requires around an hour to compute the join of a million records while ours is much more efficient.

[MRR20] showed the first practical approach for computing database joins with linear overhead in communication and computation requiring only constant rounds of communication. Their join computations are composable; the output of a join can serve as the input for the next join operation enabling a sequence of SQL join computations, all while hiding the size of the each intermediate joins output. In terms of technique, this approach shows how to combine the binary-secret sharing scheme of [MR18b] with cuckoo hashing (a common feature of most efficient and state of the art approaches for PSI [PSZ14b, PSSZ15b, KKRT16, PSTY19a]). To compute an oblivious shuffle they design a custom protocol for evaluating the oblivious switching network in the three party setting. The main drawback of their approach is that they require the primary keys of the both the input tables to be *unique*. This assumption limits the protocol from being truly composable for real world datasets. In this work, we are able to overcome this limitation provided it is safe to *leak the size of the output table*.

Secrecy. Liagouris et al. in [LKFV21] proposed a generic MPC-based framework to handle the whole suite of SQL query operations. They have secret-shared inputs anddd outputs and can handle non-unique join keys. To compute an inner join of two input tables of size n, they require $O(n^2)$ communication and constant rounds. They achieve constant round by doing a comparison between all pairs of join keys in parallel. Since, they assume the worst case upper bound on the output table, they are able to hide the intermediate output table size. In comparison, our work takes an oblivious sorting and the novel aggregation trees based approach to reduce the asymptotic communication to $O((n+D)\log n)$ in $O(\log n)$ rounds, where D is the output table size. As such, our protocols significantly outperform Secrecy when one table has unique keys or if $D \ll n^2/\log n$. As the output table size approaches n^2, the additional complexity of our protocol will result in worse performance. The exact trade-off point is unclear but certainly for most practical applications with large n and moderate D, e.g. $n \approx D \approx 2^{20}$, our protocol is orders of magnitude faster.

In terms of setting, Secrecy can be easily generalized to any semi-honest setting as long as we can efficiently instantiate the MPC ideal functionality; our work can also be generalized to any semi-honest setting by instantiating the MPC ideal functionality and oblivious sorting in the chosen setting.

Privately computing set operations has been widely studied in recent years, with much focus on the computing the intersection of sets [PSZ14b, PSSZ15a, PSZ16, KKRT16, CLR17, CHLR18, IKN$^+$17a, RA18, PRTY19, PRTY20]. Most of these works are in the two party setting and not amenable secret-shared inputs. Some works in the two party setting [PSSZ15a, PSWW18, CO18, PSTY19b, GPR$^+$21, RS21, RR22] look at revealing secret-shares of the output to enable computing over the intersection.

3 Preliminaries

Notation. We use $x := y$ to [re]define the variable x with the value of y. $x = y$ denotes mathematical equality or the bit b which is 1 iff x, y hold the same value. Let $[m, n]$ denote the "inclusive" range $\{m, m{+}1, 2, ..., n\}$ and $[n]$ as shorthand for $[1, n]$. We also define $(m, n]$ as the "left-exclusive" range $\{m+1, 2, ..., n\}$, $[m, n)$ as the "right-exclusive" $\{m, m+1, 2, ..., n-1\}$ and $(n], [n)$ as the shorthands for $(1, n], [1, n)$, respectively. Let V be a vector with elements $V = (V_1, ..., V_n)$. A subvector can be indexed using $V_{[m,n]}$ to denote $(V_m, ..., V_n)$. For a matrix M we denote the ith

row as M_i and the jth column as $M_{*,j}$. The element in row i and column j is indexed as $M_{i,j}$. We also denote a submatrix by sub-scripting it with the row/column set. Let $\|$ and $//$ denote the horizontal and vertical concatenation of two matrices, respectively.

Typically X, Y will refer to the input tables that are being joined. We use n to represent the number of rows a table has, which for simplicity will assume to be the same for X, Y.

We define a permutation of size m as an bijective function $\pi : [m] \rightarrow [m]$. We extend this definition such that when π is applied to a vector V of m elements, then $\pi(V) = (V_{\pi(1)}, ..., V_{\pi(m)})$. Parties are referred to as P_0, P_1, P_2. We use κ to denote the computational security parameter, e.g. $\kappa = 128$, and λ as the statistical security parameters, e.g. $\lambda = 40$.

Let $[\![x]\!]$ refer to a binary secret sharing of x. For example, $x \in \{0, 1\}^m$ could be a bit string while $[\![x]\!]$ is a binary secret sharing between the parties. We extend this notation to vectors and other structured objects in the natural way, e.g. $[\![V]\!]$ is the sharing of vector $V \in \mathbb{G}^n$ for $\mathbb{G} = \{0, 1\}^m$. The share held by party P_i is denoted as $[\![x]\!]_i$. We use the notation $[\![x]\!] \in \mathbb{G}$ to denote that x is an element of $\mathbb{G}$ and $[\![x]\!]$ is a sharing of it. We also make use of arithmetic (over an appropriate field) secret shares $\langle\!\langle x \rangle\!\rangle$ and secret shares $(\![\pi]\!)$ of a permutation π.

3.1 Secure Computation Framework

Our protocols are general and can be instantiated in any setting that implements our ideal MPC functionality $\mathcal{F}_{\mathsf{mpc}}$. For ease of presentation we will assume two basic types of secret shares: binary shares denoted as $[\![x]\!]$, and arithmetic sharing scheme denoted as $\langle\!\langle x \rangle\!\rangle$. For $\langle\!\langle x \rangle\!\rangle$, we will require the domain to be large enough to hold an index of size $O(n)$, e.g. 32-bit shares should suffice. We leave the exact specification of these share types as an implementation detail. We will however assume binary and arithmetic shares can be XOR'ed and added non-interactively, respectively. Additionally, we require the distribution of the corrupt parties shares to be simulatable, i.e., independent of the under laying value.

We define $\mathcal{F}_{\mathsf{mpc}}$ in Figure 1 which takes a circuit $\mathcal{C}$ and secret shares $[\![x]\!]_i$ as inputs, and outputs new shares $[\![y]\!]_i$, where y is the output of the circuit $\mathcal{C}$ applied to the reconstructed input x.We assume the functionality can take any type of secret share input. Moreover, let $\mathcal{F}_{\mathsf{conv}}$ be the functionality that can convert between the $[\![x]\!]$,$\langle\!\langle x \rangle\!\rangle$ share representation, see [MR18a] for an example in the three party setting.

When discussing security, we assume a simplified UC model. Conceptually, the functionalities are performed by an idealized trusted third party that faithfully performed the desired computation and returns the result to each of the parties. All parties have a secure and private channel with the ideal functionalities. In the semi-honest setting, we say a protocol is secure if the protocol is correct and messages received by the corrupt parties could have be simulated from just their final output. We refer to [Lin16] for a more detailed description.

3.2 Secret-shared Sorting

In our protocols we need a sorting functionality in the secret-shared setting that implements the ideal sorting functionality $\mathcal{F}_{\mathsf{sort}}$ illustrated in Figure 2. The sorting functionality outputs a secret shared permutation $(\![\pi]\!)$ that can be applied to a secret shared vector to permute the vector using $\mathcal{F}_{\mathsf{perm}}$ functionality as presented in Figure 3. One could implement $\mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Perm}}$ using $\mathcal{F}_{\mathsf{MPC}}$ but often there are more efficient protocols, e.g. [CHI+19a].

PARAMETERS: Participating parties $P_1, ..., P_{m-1}$. Circuit $\mathcal{C}$ encodes the functionality to be computed.

FUNCTIONALITY: Upon command $(\mathcal{C}, [\![x]\!]_i)$ from each P_i

- Reconstruct inputs $x := \mathsf{reconstruct}([\![x]\!])$
- Compute $[\![y]\!] := \mathsf{share}(\mathcal{C}(x))$
- Send $[\![y]\!]_i$ to P_i

Figure 1: Secure Computation ideal functionality $\mathcal{F}_{\mathsf{mpc}}$

Upon command $(\textsc{Sort}, [\![K]\!]_i)$ from $P_i \in \{P_0, \cdots, P_{m-1}\}$

- Reconstruct $K := \mathsf{reconstruct}([\![K]\!])$.
- Let $\pi : [n] \to [n]$ be the unique permutation such that $(K_{\pi(i)}, \pi(i)) < (K_{\pi(i')}, \pi(i'))$ for all $i, i' \in [n]$ where $i < i'$, i.e. π is the stable sort permutation.
- Output $(\![\pi]\!)_i$ to the party P_i where $(\![\pi]\!) := \mathsf{PermShare}(\pi)$.

Figure 2: Secret-shared Sorting ideal functionality $\mathcal{F}_{\mathsf{sort}}$

3.3 Implementation

When we implement our protocols we realize $\mathcal{F}_{\mathsf{MPC}}$ with the honest majority three party protocol of [MR18a]. Binary and arithmetic shares are implemented using replicated shares, i.e. $[\![x]\!]$ is split into three random values x_1, x_2, x_3 s.t. $x = x_1 \oplus x_2 \oplus x_3$ and each of the three parties hold a differ set two of the x_i values. $(\![x]\!)$ is defined similarly.

For $\mathcal{F}_{\mathsf{Sort}}$ and $\mathcal{F}_{\mathsf{Perm}}$ we make use of the recent efficient construction for secret-shared sorting protocol from [CHI+19a]. Other setting, e.g. two party, could also be considered. However, we chose to implement the three party honest majority setting due to the concrete efficiencies it offers. We note that [CHI+19a] could be implemented in the two party setting with the same asymptotics, i.e. $O(\log n)$ rounds and $O(n \log n)$ time.

Upon command $(t, (\![\pi]\!)_i, [\![V]\!]_i)$ from $P_i \in \{P_0, \cdots, P_{m-1}\}$

- Reconstruct $V := \mathsf{reconstruct}([\![V]\!]), \pi := \mathsf{permReconstruct}((\![\pi]\!))$.
- If $t = \textsc{Perm}$, output $[\![V']\!]_i$ to the party P_i where $[\![V']\!] := \mathsf{share}(\pi(V))$.
- If $t = \textsc{InvPerm}$, output $[\![V']\!]_i$ to the party P_i where $[\![V']\!] := \mathsf{share}(\pi^{-1}(V))$.

Figure 3: Secret-shared permutation ideal functionality $\mathcal{F}_{\mathsf{Perm}}$

4 Technical Overview

Aggregation Tree. To build our join protocols, we introduce our novel *aggregation trees* construction. We present three variants, prefix, suffix and full aggregation. The construction is parameterized by a binary operator $\star : \mathbb{D}^2 \to \mathbb{D}$, an input vector $[\![\mathcal{V}]\!] \in \mathbb{D}^n$ and a vector of *control bits* $[\![\mathcal{B}]\!] \in \{0,1\}^n$. For example, $\star$ can be thought of as integer addition. The control bits logically define divisions of $\mathcal{V}$ into sub-vectors, or *blocks*. $\mathcal{B}_i = 0$ indicates that index i is the start of a block.

For prefix aggregation, the output $[\![\mathcal{V}']\!]$ can then be defined as a left to right pass over each block where $\mathcal{V}'_i := \mathcal{V}'_i$ if i is the start of a block and $\mathcal{V}'_i := \mathcal{V}_j \star ... \star \mathcal{V}_{i-1} \star \mathcal{V}_i$ otherwise, where j is the start of the block containing i. Suffix aggregation is defined in the same way except in a right to left manner, i.e. $\mathcal{V}_i = \mathcal{V}_i \star ... \star \mathcal{V}_j$ where j is the end of a block. Note that the definition of a block has not changed. Finally, full aggregation can be defined as $\mathcal{V}_i = \mathcal{V}_j \star ... \star \mathcal{V}_i \star ... \star \mathcal{V}_{j'}$ where j, j' is the start and end of the block containing i.

A naive method for implementing the aggregation functionality would be to perform a left and/or right linear pass over $\mathcal{V}$ to compute $\mathcal{V}'$. However, this would be impractical due to it requiring $O(n)$ rounds. Instead we propose a protocol framework that allows the aggregation to be computed in $O(\log n)$ rounds and $O(n)$ time. Intuitively, this improvement is achieved by changing the order in which the $\star$ operator is applied. To facilitate this, we will require $\star$ to be associative, i.e., $(a \star b) \star c = a \star (b \star c)$ for all $a, b, c \in \mathbb{D}$. We aggregate using a binary tree structure and assign the vector $\mathcal{V}$ to the leaves of the binary tree. We have an *upstream* phase, where values at the leaves are propagated up to the root node followed by a *downstream* phase where values are propagate back to the leaves to obtain $\mathcal{V}'$. Critically, all values computed in this process require only the current parent and two children nodes and as such can be performed in $O(\log n)$ rounds in a level by level manner in our binary tree. We apply this aggregating technique for different tasks in our join protocols. For each, we instantiate the tree by defining the $\star$ operator and then performing prefix, suffix or full aggregation.

Secure Join. In [MRR20], the authors presented an $O(n)$ computation and communication protocol $\Pi_{\mathsf{Join\text{-}OO}}$ in constant-rounds to compute joins over secret-shared input database tables. The major drawback of their approach is that the join required *unique* primary keys, i.e., each row is join in a one-to-one manner. As discussed earlier, it is non-trivial to bypass this requirement with existing approaches that are competitive. We introduce a novel approach that allows one or both of the two inputs tables to have *non-unique* primary keys with *unbounded repeats*. We present two join protocols $\Pi_{\mathsf{Join\text{-}OM}}$ and $\Pi_{\mathsf{Join\text{-}MM}}$; with the first being optimized for the case where one table has unique keys, i.e., each row in the unique table has a one-to-many relation with rows in the other table. This protocol requires $O(n \log n)$ computation and $O(\log n)$ rounds. The $\Pi_{\mathsf{Join\text{-}MM}}$ protocol is more complicated due to the possibility for having an output size of $m = n^2$ in the case that both tables consists of n copies of a single key. To get practical efficiency we provide a mechanism to reveal an upper bound D on the size of the output table m, e.g. D as the next power of two of m or a differentially private upper bound of m. The $\Pi_{\mathsf{Join\text{-}MM}}$ protocol then has running time $O((n + D) \log n)$ and takes $O(\log n)$ rounds. The protocols are very general and can be modified to compute a variety of related functionalities. For example, it is a simple task to extend our protocol to compute full joins or unions.

Our protocols make use of the ability to sort a secret shared list. While sorting certainly makes the task simpler by bring matching records next to each other, it remains non-trivial to match up

the correct records in an oblivious manner. For example, a records might be matched one-to-one, one-to-many or many-to-many. The protocol must be oblivious to all of these possibilities.

One-to-Many Joins $\Pi_{\textsf{Join-OM}}$. Let us assume X is the table with unique primary keys. First we will combine the two tables as

$$Z := \begin{bmatrix} \mathsf{key}(X) & X & 0 \\ \mathsf{key}(Y) & 0 & Y \end{bmatrix}$$

where $\mathsf{key}(X), \mathsf{key}(Y)$ is the join key columns of X and Y respectively, see Appendix A for details. The first step of our protocol is to sort the rows of Z by the join key. We will stable sort so that the row X_i to appear before any matching rows from Y. In particular, if X_i matches rows $Y_{j_1}, ..., Y_{j_t}$ then they will be some index ℓ s.t. $Z_\ell = [\mathsf{key}(X_i), \quad X_i, \quad 0]$ and $Z_{\ell+k} = [\mathsf{key}(Y_{j_k}), \quad 0, \quad Y_{j_k}]$ for $k \in [t]$. It is then the task to populate a row of the output table for each pair $[X_i, Y_{j_1}], ..., [X_i, Y_{j_t}]$. Conceptually this can be done by simply copying X_i into the next t rows.

The first challenge in achieving this is to obliviously decide when an X_i should be copied onto the next row. This can be done by comparing the key for the current row Z_ℓ with the join key for the next row $Z_{\ell+1}$. If they are equal then the Z_ℓ row's X record should be copied into $Z_{\ell+1}$.

Continuing the example above, $\mathsf{key}(X_i)$ will match $\mathsf{key}(Y_{j_1})$ and therefore X_i contained in row Z_ℓ should be copied into row $Z_{\ell+1}$. We will also have the key for row $Z_{\ell+1}$, i.e. $\mathsf{key}(Y_{j_1})$, match with the key of row $Z_{\ell+2}$, i.e. $\mathsf{key}(Y_{j_2})$, and as such X record in row $Z_{\ell+1}$ will be copied into row $Z_{\ell+2}$. After the copies are performed, we will have $Z_\ell = [\mathsf{key}(X_i), \; X_i, \; 0]$ and $Z_{\ell+k} = [\mathsf{key}(Y_{j_k}), \; X_i, \; Y_{j_k}]$. All unmatched rows and the $Z_\ell = [\mathsf{key}(X_i), \quad X_i, \quad 0]$ rows are marked as null.

At first glance it would appear that this strategy would require the computation to perform a "left to right linear" pass over the rows of Z where each Z_ℓ row's X record is copied into $Z_{\ell+1}$ if they have matching keys. Naively this would require $O(n)$ rounds of interaction. However, based on our introduction of aggregation trees it should not be too difficult to see we can apply them here and perform the copying in $O(\log n)$ rounds as opposed to linear. In particular, we can define the control bits as $\mathcal{B}_{\ell+1} := (\mathsf{key}(Z_\ell) \stackrel{?}{=} \mathsf{key}(Z_{\ell+1}))$. Then we will define $\star$ as $\star(x_0, x_1) := x_0$. This logically results in the desired behavior, a left to right linear pass where the previous value is copied if the keys are equal.

Many-to-Many Joins $\Pi_{\textsf{Join-MM}}$. Now we overview our construction for joining tables that both have unbounded multiplicity. At first glance this task appears significantly more challenging than the one-to-many joins due to the need to effectively "multiply" sets of rows together. In particular, let $S(X,k) := \{X_i \mid \mathsf{key}(X_i) = k\}$ denote the set of rows in X that have key k. Then the task of a many-to-many join is to multiply the sets $S(X,k) \cdot S(Y,k) = \{x \| y \mid x \in S(X,k), y \in S(Y,k)\}$ for all k in the two table. For example, if $|S(X,k)| = 4$ and $|S(Y,k)| = 3$, then $|S(X,k) \cdot S(Y,k)| = 12$. In total the output table will have size $D := \sum_{k \in K} |S(X,k) \cdot S(Y,k)|$ where K is the set of keys contained in the two tables. While achieving this in an oblivious manner is non-trivial, we in fact already have the necessary building blocks.

At a high level the protocol proceeds by sorting X, Y together. If some row X_i is matched with $m = |S(Y, \mathsf{key}(X_i))|$ rows from Y, then the X_i row is duplicated m times. Similarly, rows from Y are duplicated the number of times that they have matching rows in X. Finally, we reorder the combined X, Y table such that each copy of a row from X is paired with a matching copy of a row from Y.

In more detail, the protocol begins by combining X, Y into $Z := \begin{bmatrix} \mathsf{key}(X) & X & 0 \\ \mathsf{key}(Y) & 0 & Y \end{bmatrix}$ and stable sorting the rows based on the keys. The next task is to determine for each Z_ℓ how many rows from X and, separately, Y have key $\mathsf{key}(Z_i)$. This is used to determine how many copies of Z_ℓ to make. In particular, each row Z_ℓ will have an associated pair $(m_\ell^{\mathsf{X}}, m_\ell^{\mathsf{Y}}) = (|S(X,k)|, |S(Y,k)|)$ where $k = \mathsf{key}(Z_\ell)$. We show how to compute $(m_\ell^{\mathsf{X}}, m_\ell^{\mathsf{Y}})$ using a full aggregation tree in Section 7.

The next task is to duplicate each row Z_ℓ the necessary number of times. If row Z_ℓ is from the X table, i.e. $Z_\ell = [\mathsf{key}(X_i), \ \ , X_i, \ \ 0]$, then it must be duplicated $m_\ell^* := m_\ell^{\mathsf{Y}}$ times. Otherwise, Z_ℓ is from Y and needs to be duplicated $m_\ell^* := m_\ell^{\mathsf{X}}$ times. The protocol proceeds by arranging for m_ℓ^* dummy/unused rows to be after row Z_ℓ. This is achieved by computing an index $q_\ell \in [D]$ for each row Z_ℓ so that if Z_ℓ was moved to row index q_ℓ, it would be followed by the m_ℓ^* unused dummy rows. The necessary dummy rows are added to the Z tables and sorted by the q_ℓ indices. The Z_ℓ rows are then duplicated m_ℓ^* times using an aggregation tree. In particular, each Z_ℓ row from X with key k is now copied $m_\ell^{\mathsf{Y}} = |S(Y,k)|$ times and each from Y with key k is copied $m_\ell^{\mathsf{X}} = |S(X,k)|$ times.

The final step in the protocol is to reorder the Z_ℓ rows to bring together the matching copies of the X, Y rows. In particular, we will reorder the "Y" columns of Z such that if X_i matches with Y_j, then one of the Z_ℓ rows containing Y_j is permuted to one of the $Z_{\ell'}$ rows containing X_i. This is achieved by computing another index $d_\ell \in [D]$ for each row Z_ℓ which encodes this mapping. In the example above, $d_\ell = \ell'$.

The protocol then permutes the rows of the "Y" columns of Z based on the d_ℓ indices. The result is that Z will now contain all of the joined rows with the correct multiplicities. Finally, any rows not corresponding to the joined result are marked as null.

We note that the full protocol is somewhat more complicated due to various details being simplified for exposition, e.g. how exactly are the indices computed. In addition, this overview can be optimized in various ways.

The Aggregate-Sort-Compute pattern We make the observation that our protocols follow a general pattern where first an aggregation is applied to compute some ordering in the form of a index value. Then we apply sorting to get the rows in the desired order. This is then followed by updating each row as a function of its immediate neighbors. By repeatedly applying this pattern, we observe that highly complex operations can be performed that traditionally would have required expensive techniques such as oblivious RAM. We argue that the generality of this pattern will make it useful in a wide variety of applications.

5 Parallel Aggregation using Trees

Consider an array of n values $\mathcal{V} = [v_1, v_2, \ldots, v_n]$ where each v_i belongs to some domain $\mathbb{D}$. Let $\star : \mathbb{D}^2 \to \mathbb{D}$ be an in-fixed operator defined on the domain $\mathbb{D}$ that is closed and associative. We define three forms of aggregation over the array $\mathcal{V}$ with respect to the control bits $\mathcal{B} = [b_1 = 0, b_2, \ldots, b_n]$, where b_i is a bit, as follows:

Definition 5.1. ***Prefix Aggregation.*** *The prefix aggregation $\mathcal{V}'$ of $\mathcal{V}$ with respect to $\mathcal{B}$ is defined as $\mathcal{V}' = [v'_1, v'_2, \ldots, v'_n]$, where*

$$v'_i = \bigstar_{j=\textsf{pre-ind}(i)}^{i} v_j$$

and $\textsf{pre-ind}(i) \in [n]$ *is defined to be the unique index* $\leq i$ *such that* $b_{\textsf{pre-ind}(i)} = 0$ *and* $b_j = 1$ *for all* $j \in (\textsf{pre-ind}(i), i]$.

For example, if $n = 5$ and $\mathcal{B} = \big[0, 0, 1, 1, 0\big]$, then $\mathcal{V}' = \big[v_1, v_2, v_2 \star v_3, v_2 \star v_3 \star v_4, v_5\big]$.

Definition 5.2. ***Suffix Aggregation.*** *The suffix aggregation* $\mathcal{V}'$ *of* $\mathcal{V}$ *with respect to* $\mathcal{B}$ *is defined as* $\mathcal{V}' = \big[v'_1, v'_2, \ldots, v'_n\big]$, *where*

$$v'_i = \bigstar_{j=i}^{\textsf{suf-ind}(i)} v_j$$

and $\textsf{suf-ind}(i) \in [n]$ *is defined to be the unique index* $\geq i$ *such that* $b_{\textsf{suf-ind}(i)+1} = 0$[1] *and* $b_j = 1$ *for all* $j \in (i, \textsf{suf-ind}(i)]$.

For example, if $\mathcal{B} = \big[0, 0, 1, 1, 0\big]$, then $\mathcal{V}' = \big[v_1, v_2 \star v_3 \star v_4, v_3 \star v_4, v_4, v_5\big]$.

Definition 5.3. ***Aggregation.*** *The aggregation* $\mathcal{V}'$ *of* $\mathcal{V}$ *with respect to* $\mathcal{B}$ *is defined as* $\mathcal{V}' = \big[v'_1, v'_2, \ldots, v'_n\big]$, *where*

$$v'_i = \bigstar_{j=\textsf{pre-ind}(i)}^{\textsf{suf-ind}(i)} v_j$$

and $\textsf{pre-ind}(i)$ *and* $\textsf{suf-ind}(i)$ *are as defined above.*

For example, if $\mathcal{B} = \big[0, 0, 1, 1, 0\big]$, then $\mathcal{V}' = \big[v_1, v_2 \star v_3 \star v_4,\ v_2 \star v_3 \star v_4,\ v_2 \star v_3 \star v_4,\ v_5\big]$.

All three aggregations described above can be computed with a single forward and/or single backward linear pass on the arrays but this requires $\mathcal{O}(n)$ rounds of computation. We describe a method for "parallel aggregation" using a binary tree structure which significantly improves the round complexity of the computation. In our binary tree, we associate each leaf with an entry of $\mathcal{V}$ (for ease of exposition, we assume that the array size n is a power of 2). The idea is to use internal nodes of the tree to move the necessary aggregated information through the array. We do this in two phases – an upstream phase where necessary aggregated information is pumped upwards from the leaves to the root, followed by a downstream phase where the necessary aggregated information is pushed to the leaves. All the internal nodes at a given level or height in the tree can perform their operations in parallel. As a result, each phase of the aggregation can be performed in $\log n$ steps and the entire aggregation can be performed in $\mathcal{O}(|\star| \log n)$ rounds where $|\star|$ is the multiplicative depth of $\star$.

5.1 Prefix Aggregation – A First Solution

In Figure 4, we present the steps of our algorithm and justify the correctness of our method.

Theorem 5.4. *Given input array* $\mathcal{V} = \big[v_1, v_2, \ldots, v_n\big]$ *with associated control bits* $\mathcal{B} = \big[b_1 = 0, b_2, \ldots, b_n\big]$, *protocol Figure 4 correctly computes the prefix aggregation* $\mathcal{V}' = \big[v'_1, v'_2, \ldots, v'_n\big]$ *where* $v'_i = \bigstar_{j=\textsf{pre-ind}(i)}^{i} v_j$ *according to definition 5.1.*

Proof. Let's assign semantics for the entries in the 5-tuple $(\alpha, \beta, \textsf{prefix}, \ell, p)$ used to represent every node in the binary tree.

[1] For consistency, we assume $b_i = 0$ for $i \notin [n]$.

PARAMETERS:

- Input: array $\mathcal{V} = [v_1, v_2, \ldots, v_n]$, each v_i belongs to some domain $\mathbb{D}$.
- Input: array $\mathcal{B} = [b_1 = 0, b_2, \ldots, b_n]$, where b_i is a bit.
- Operator: $\star : \mathbb{D}^2 \to \mathbb{D}$ is closed and associative on $\mathbb{D}$. $\mathbb{D}$ has identity element $e_\star$ with respect to $\star$.
- Binary tree $\mathcal{T}$ where each node is represented by a 5-tuple $(\alpha, \beta, \mathsf{prefix}, \ell, p)$.

PROTOCOL: Upon input (t = $\mathsf{Prefix\ Aggregation}$, $\star$, $\mathcal{V}$, $\mathcal{B}$)

- *Initialization.* Each leaf is assigned $(e_\star, v_i, e_\star, b_i, b_i)$.
- *Upstream* For every internal node we assign $(\alpha, \beta, \mathsf{prefix}, \ell, p)$ as

$$\alpha := \beta_0$$
$$\beta := \begin{cases} \beta_1 & p_1 = 0 \\ \beta_0 \star \beta_1 & p_1 = 1 \end{cases}$$
$$\mathsf{prefix} := e_\star, \ell := \ell_0, p := p_0 p_1$$

  where $(\alpha_0, \beta_0, \mathsf{prefix}_0, \ell_0, p_0)$ is left child and $(\alpha_1, \beta_1, \mathsf{prefix}_1, \ell_1, p_1)$ is right child of internal node.

- *Downstream* We calculate prefix values of left child prefix_0 and right child prefix_1 as

$$\mathsf{prefix}_0 = \mathsf{prefix}$$
$$\mathsf{prefix}_1 = \begin{cases} e_\star & \ell_1 = 0 \\ \alpha & \ell_1 = 1, p_0 = 0 \\ \mathsf{prefix} \star \alpha & \ell_1 = 1, p_0 = 1 \end{cases}$$

  of every internal node $(\alpha, \beta, \mathsf{prefix}, \ell, p)$.

- *Output*: Aggregation array $\mathcal{V}_{\text{pre}} = [v'_1, v'_2, \ldots, v'_n]$ where $v'_i = \mathsf{prefix}_i \star v_i$ and prefix_i is the prefix value of leaf v_i during downstream.

PROTOCOL: Upon input (t = $\mathsf{Suffix\ Aggregation}$, $\star$, $\mathcal{V}$, $\mathcal{B}$)

- *Initialization.* Each leaf is assigned $(e_\star, \delta_i, e_\star, b_i, b_i)$ where $\delta_i = \begin{cases} e_\star & b_i = 0 \\ v_i & b_i = 1 \end{cases}$
- *Upstream* For every internal node we assign

$$\gamma = \delta_1$$
$$\delta = \begin{cases} \delta_0 & p_0 = 0 \\ \delta_0 \star \delta_1 & p_0 = 1 \end{cases}$$
$$\mathsf{suffix} = e_\star, \ell = \ell_0, p = p_0 p_1$$

  where $(\gamma_0, \delta_0, \mathsf{suffix}_0, \ell_0, p_0)$ is left child and $(\gamma_1, \delta_1, \mathsf{suffix}_1, \ell_1, p_1)$ is right child of internal node.

- *Downstream* We calculate suffix values of left child suffix_0 and right child suffix_1 as

$$\mathsf{suffix}_0 = \begin{cases} \gamma & p_1 = 0 \\ \gamma \star \mathsf{suffix} & p_1 = 1 \end{cases}$$
$$\mathsf{suffix}_1 = \mathsf{suffix}$$

  of every internal node $(\gamma, \delta, \mathsf{suffix}, \ell, p)$.

- *Output*: Aggregation array $\mathcal{V}_{\text{suf}} = [v'_1, v'_2, \ldots, v'_n]$ where $v'_i = v_i \star \mathsf{suffix}_i$ and suffix_i is the suffix value of the leaf v_i at the end of downstream.

Figure 4: Aggregation Algorithms using Binary Tree.

- β is the sum of the leaves of the sub-tree rooted at the internal node, starting from the right until and including the first value whose corresponding control bit is 0 (if it exists). More formally,

$$\beta = \bigstar_{j=\max\{i_{\text{left}},\textsf{pre-ind}(i_{\text{right}})\}}^{i_{\text{right}}} v_j$$

  where i_{left} and i_{right} are the indices of the left-most and right-most leaves of the sub-tree rooted at the internal node respectively.

- α is the β value of the left sub-tree of the sub-tree rooted at the internal node.

- prefix is the prefix aggregation value of the left-most leaf of the sub-tree rooted at the internal node, excluding the value of the left-most leaf. More formally,

$$\textsf{prefix} = \bigstar_{j=\textsf{pre-ind}(i_{\text{left}})}^{i_{\text{left}}-1} v_j$$

  where i_{left} is the index of the left-most leaf of the sub-tree rooted at the internal node.

- ℓ is the control bit of the left-most leaf in the sub-tree rooted at the internal node.

- p is the product (logical conjunction) of the control bits of all the leaves of the sub-tree rooted at the internal node.

Initialization. By definition, the β values at the leaves are the values in $\mathcal{V}$. Since leaves have empty left sub-trees, the α value of a leaf is $e_\star$. By definition, the ℓ and p values at the leaves are the control bits $\mathcal{B}$. The prefix values are only calculated during the downstream phase and throughout the upstream phase, we simply set $\textsf{prefix} = e_\star$. Therefore, the 5-tuples at the leaves are consistent with the above semantics.

Upstream. During the upstream, the update to α as $\alpha = \beta_0$ is consistent by definition. Since the prefix is only calculated during the downstream phase we simply set $\textsf{prefix} = e_\star$. The left-most leaf of the left sub-tree rooted at a node is the same as the left-most leaf of the sub-tree rooted at the same node, the update $\ell = \ell_0$ is consistent. It is also easy to see that the product of the control bits of all the leaves of the sub-tree rooted at a node is the product of control bits from the sub-trees rooted at the left and right child, the update $p = p_0 p_1$ is consistent. Now, we look at the update to β. Recall that β is the sum of the leaves of the sub-tree rooted at the internal node, starting from the right until and including the first value whose corresponding control bit is 0 (if it exists). It is helpful to think of β as the "trailing sum" (sum of values associated with the right-most sequence of control bits of the form $(0)1111\ldots1)$ of the sub-tree rooted at the internal node. If the right sub-tree of an internal node contains a leaf with the control bit 0, then the trailing sum of the sub-tree rooted at the internal node is the same as the trailing sum of the right sub-tree of the node. So, if $p_1 = 0$ (the product of the control bits in the right sub-tree is 0 if and only if one of the control bits is 0), then $\beta = \beta_1$. In the other case, when $p_1 = 1$ (the right sub-tree has leaves all with control bits 1), β_1 the trailing sum would be the aggregate of β_1 and the trailing sum of the left sub-tree. So if $p_1 = 1$, then $\beta = \beta_0 \star \beta_1$. Thus, in the upstream, we update the 5-tuples in a way that is consistent with our semantics.

Downstream. Next, we show how the prefix values are updated correctly during the downstream phase. At the root, the prefix value is the prefix aggregation value of the left-most leaf of

the entire tree, excluding its own value. This amounts to any empty set of leaves and $\mathsf{prefix} = \bigstar_{j=\mathsf{pre\text{-}ind}(0)=0}^{0-1} v_j = e_\star$. For some internal node, let's verify if the prefix values of the left and right child are updated correctly. Since the left-most leaf of the left sub-tree of a node is the same as the left-most leaf of the sub-tree rooted at the same node $\mathsf{prefix}_0 = \mathsf{prefix}$. For the right sub-tree, if $l_1 = 0$ (the left-most leaf of the right sub-tree has control bit 0), then the prefix aggregation value is $e_\star$ by definition. If the left-most leaf of the right sub-tree of the node had the control bit 1, then the prefix aggregation value will depend on the left sub-tree of the node. If the left sub-tree has atleast one leaf with the control bit 0, then the prefix aggregation value of the left-most leaf of the right sub-tree of the node would simply be the trailing sum of the left sub-tree of the node. Formally, this means that if $\ell_1 = 1$ and $p_0 = 0$ (the product of the control bits in the left sub-tree is 0 if and only if one of the control bits is 0), then $\mathsf{prefix}_1 = \alpha$. If the left sub-tree had no leaf with the control bit 0, then the prefix aggregation value of the left-most leaf of the right sub-tree of the node would be the aggregation of the leaves of the left sub-tree of the node as well as the prefix-aggregation value of the left-most leaf of the left sub-tree of the node. Formally, this means that if $\ell_1 = 1$ and $p_0 = 1$, then $\mathsf{prefix}_1 = \mathsf{prefix} \star \alpha$. Thus, we have shown inductively that the prefix values are updated correctly during the downstream phase and are consistent with the semantics described above.

Output. Finally, we show that $\mathcal{V}_{\text{pre}}$ is calculated correctly. Indeed, by our previous arguments,

$$\mathsf{prefix}_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{i-1} v_j$$

Hence

$$v'_i = \mathsf{prefix}_i \star v_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{i} v_j$$

as required. □

5.2 Suffix Aggregation – A First Solution

Again, in Figure 4, we present the steps of our algorithm and justify the correctness of our method.

Theorem 5.5. *Given input array $\mathcal{V} = [v_1, v_2, \ldots, v_n]$ with associated control bits $\mathcal{B} = [b_1 = 0, b_2, \ldots, b_n]$, protocol Figure 4 correctly computes the suffix aggregation $\mathcal{V}' = [v'_1, v'_2, \ldots, v'_n]$ where $v'_i = \bigstar_{j=i}^{\mathsf{suf\text{-}ind}(i)} v_j$ according to definition 5.2.*

Proof. We defer the proof to Section B since it is similar to our reasoning for prefix aggregation.

5.3 Full Aggregation – A First Solution

Our algorithm simply runs the prefix and suffix aggregation algorithms in parallel and combines the results from the two at the very end. For each node in the tree, we define an 8-tuple $(\alpha, \beta, \gamma, \delta, \mathsf{prefix}, \mathsf{suffix}, \ell, p)$. We follow the steps of Figure 4 to run both prefix and suffix aggregation simultaneously and update the 8-tuple.

Theorem 5.6. *Given input array $\mathcal{V} = [v_1, v_2, \ldots, v_n]$ with associated control bits $\mathcal{B} = [b_1 = 0, b_2, \ldots, b_n]$, protocol Figure 4 correctly computes the suffix aggregation $\mathcal{V}' = [v'_1, v'_2, \ldots, v'_n]$ where $v'_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{\mathsf{suf\text{-}ind}(i)} v_j$ according to definition 5.3.*

Proof. It is straightforward to see that combining the arguments from before demonstrates the correctness of the algorithm above.

PARAMETERS: Participating parties $\{P_0, \cdots, P_{m-1}\}$ Input vector $[\![\mathcal{V}]\!] \in \mathbb{D}^n$ and control bits $[\![\mathcal{B}]\!] \in \{0,1\}^n$. An aggregation type of $t \in \{\textsc{Prefix}, \textsc{Suffix}, \textsc{Full}\}$. A user defined circuit $\star : \mathbb{D}^2 \to \mathbb{D}$ that is associative.

FUNCTIONALITY: Upon input $(t, \star, [\![\mathcal{V}]\!], [\![\mathcal{B}]\!])$ from the parties, defined $\textsf{pre-ind}(i) \in [i]$ to be the maximum value such that $\mathcal{B}_{\textsf{pre-ind}(i)} = 0$. Define $\textsf{suf-ind}(i) \in [i, n]$ to be the minimum value such that $\textsf{suf-ind}(i) = n$ or $\mathcal{B}_{\textsf{suf-ind}(i)+1} = 0$.

1. If $t = \textsc{Prefix}$, output $[\![\mathcal{V}']\!] \leftarrow \textsf{share}(\mathcal{V}')$ where $\mathcal{V}'_i := \bigstar_{j=\textsf{pre-ind}(i)}^{i} \mathcal{V}_j$
2. If $t = \textsc{Suffix}$, output $[\![\mathcal{V}']\!] \leftarrow \textsf{share}(\mathcal{V}')$ where $\mathcal{V}'_i := \bigstar_{j=i}^{\textsf{suf-ind}(i)} \mathcal{V}_j$
3. If $t = \textsc{Full}$, output $[\![\mathcal{V}']\!] \leftarrow \textsf{share}(\mathcal{V}')$ where $\mathcal{V}'_i := \bigstar_{j=\textsf{pre-ind}(i)}^{\textsf{suf-ind}(i)} \mathcal{V}_j$

Figure 5: Functionality $\mathcal{F}_{\textsf{Agg}}$ for secret shared aggregation.

5.4 Optimizations and Improvements

Let us look at our first solution to the prefix aggregation problem. Notice that β is only used in the upstream phase and prefix is only used in the downstream phase. So, we can in fact use the same variable for the two of them, only that they would have different semantics in the two phases. Furthermore, since α is always set to be β_0, it is possible to combine the two of these as well since this means that the values that really matter are the βs. Using these optimizations, we present the following improved algorithm for prefix aggregation. For each node in the tree, we we define a 2-tuple (β, p).

Initialization. We initialize the 2-tuples for the leaves as (v_i, b_i).

Upstream. Consider an internal node with left and right children having the associated 2-tuples (β_0, p_0) and (β_1, p_1) respectively. Then, we calculate the 2-tuple of the internal node as

$$\beta = \begin{cases} \beta_1 & p_1 = 0 \\ \beta_0 \star \beta_1 & p_1 = 1 \end{cases}$$

$$p = p_0 p_1$$

Downstream. Consider an internal node with an associated 2-tuple (β, p). Then, we update the β values for its left and right children as

$$\beta_1 = \begin{cases} \beta_0 & p_0 = 0 \\ \beta \star \beta_0 & p_0 = 1 \end{cases}$$

$$\beta_0 = \beta$$

Notice that the updates must be performed in the order mentioned above as the update to β_1 uses the value of β_0 (prior to its own update).

Output. We calculate the prefix aggregation of $\mathcal{V}$ as $\mathcal{V}_{\text{pre}} = \left[v'_1, v'_2, \ldots, v'_n\right]$, where

$$v'_i = \begin{cases} v_i & b_i = 0 \\ \beta_i \star v_i & b_i = 1 \end{cases}$$

and β_i is the β value in the 2-tuple associated with the leaf with value v_i and control bit b_i at the end of the downstream phase.

Proof of correctness. To see that the above algorithm indeed performs prefix aggregation, we set up the following semantics for each of the entries in the 5-tuple. For an internal node with an associated 2-tuple (β, p),

- β has different semantics during the upstream and downstream phases. During the upstream phase, β is the sum of the leaves of the sub-tree rooted at the internal node, starting from the right until and including the first value whose corresponding control bit is 0 (if it exists). More formally,

  $$\beta = \bigstar_{j=\max\{i_{\text{left}}, \mathsf{pre\text{-}ind}(i_{\text{right}})\}}^{i_{\text{right}}} v_j$$

  where i_{left} and i_{right} are the indices of the left-most and right-most leaves of the sub-tree rooted at the internal node respectively.

  During the downstream phase, β is the prefix aggregation value of the leaf preceding the left-most leaf of the sub-tree rooted at the internal node, including the value of the leaf preceding the left-most leaf. More formally,

  $$\beta = \bigstar_{j=\mathsf{pre\text{-}ind}(i_{\text{left}}-1)}^{i_{\text{left}}-1} v_j$$

  where i_{left} is the index of the left-most leaf of the sub-tree rooted at the internal node. A caveat is that the above semantic holds as long as $i_{\text{left}} > 1$.

- p is the product (logical conjunction) of the control bits of all the leaves of the sub-tree rooted at the internal node.

Initialization. It is easy to see that by definition, during the upstream phase, the β values at the leaves are the values in $\mathcal{V}$. By definition, the p values at the leaves are the control bits $\mathcal{B}$. This shows that the 2-tuples at the leaves are consistent with the above semantics.

Upstream. We now inductively show that the 2-tuples are updated correctly during the upstream phase. It is easy to see that the product of the control bits of all the leaves of the sub-tree rooted at a node is the product of those in the left sub-tree rooted at the same node multiplied by the product of those in the right sub-tree rooted at the same node, and hence the update $p = p_0 p_1$ is consistent. Now, we look at the update to β. Recall that during the upstream phase, β is the sum of the leaves of the sub-tree rooted at the internal node, starting from the right until and including the first value whose corresponding control bit is 0 (if it exists). As before, it is helpful to think of β during the upstream phase as the "trailing sum" (sum of values associated with the right-most sequence of control bits of the form $(0)1111\ldots1$) of the sub-tree rooted at the internal node. It is easy to see that if the right sub-tree of an internal node contains a leaf with the control bit 0, then the trailing sum of the sub-tree rooted at the internal node is the same as the trailing sum of the

right sub-tree of the node. Formally, this means that if $p_1 = 0$ (the product of the control bits in the right sub-tree is 0 if and only if one of the control bits is 0), then $\beta = \beta_1$. In the other case, where the right sub-tree has leaves all of whose control bits are 1, β_1, the trailing sum of the right sub-tree, would be the sum of the values of all the leaves in the right sub-tree, and the trailing sum of the entire sub-tree rooted at the internal node would be the aggregate of β_1 and the trailing sum of the left sub-tree. Formally, this means that if $p_1 = 1$, then $\beta = \beta_0 \star \beta_1$. Thus, we have shown inductively that the 2-tuples are updated correctly during the upstream phase and are consistent with the semantics described above.

Downstream. What remains to be shown is that inductively the β values are updated correctly during the downstream phase. At the root, however, $i_{\text{left}} = 1$ and hence the semantic for β that we described may not hold. Hence, our inductive hypothesis begins at the right child of the root (notice that the left child of the root also has $i_{\text{left}} = 1$). At the right child of the root, by our semantic, β must be updated to be the prefix aggregation value of the leaf preceding the left-most leaf of the sub-tree rooted at the internal node, including the value of the leaf preceding the left-most leaf. Plugging in for the variables, at the right child of the root,

$$\beta = \bigstar_{j=\mathsf{pre\text{-}ind}(i_{\text{left}}-1)}^{i_{\text{left}}-1} v_j$$

where i_{left} is the index of the left-most leaf of the sub-tree rooted at the right child of the root. But notice that $i_{\text{left}} - 1$ of the right child of the root is i_{right} of the left child of the root. Hence,

$$\bigstar_{j=\mathsf{pre\text{-}ind}(i_{\text{left}}-1)}^{i_{\text{left}}-1} v_j = \bigstar_{j=\mathsf{pre\text{-}ind}(i_{\text{right}})}^{i_{\text{right}}} v_j$$

where i_{right} is the index of the right-most leaf of the sub-tree rooted at the left child of the root. Futhermore, since i_{left} of the left child of the root is 1, and $\max\{1, \mathsf{pre\text{-}ind}(i_{\text{right}})\} = \mathsf{pre\text{-}ind}(i_{\text{right}})$, what we have is that β of the right child of the root must be updated to the value β_0 of the left child of the root that was computed during the upstream phase. Indeed, as per our update rules, β of the right child of the root has been correctly set, establishing the base case of our induction. We now proceed with the inductive claim. Suppose we have the 2-tuple at an internal node. We would like to update the β values of the left and right children of this node and proceed inductively. Proceeding in order, let us first consider the right sub-tree of the node. The prefix aggregation value of the leaf preceding the left-most leaf of this right sub-tree (which is the right-most leaf of the left sub-tree of the node), including its own value, would include the trailing sum of the left sub-tree of the node. If the left sub-tree had a leaf with the control bit 0, then the prefix aggregation value of the leaf preceding the left-most leaf of the right sub-tree of the node would simply be the trailing sum of the left sub-tree of the node. Formally, this means that if $p_0 = 0$ (the product of the control bits in the left sub-tree is 0 if and only if one of the control bits is 0), then $\beta_1 = \beta_0$. If the left sub-tree had no leaf with the control bit 0, then the prefix aggregation value of the leaf preceding the left-most leaf of the right sub-tree of the node would be the aggregation of the leaves of the left sub-tree of the node as well as the prefix-aggregation value of the leaf preceding the left-most leaf of the left sub-tree of the node. Formally, this means that if $p_0 = 1$, then $\beta_1 = \beta \star \beta_0$. If the left-most leaf of the left sub-tree of the node has index 1, then β would not have the right semantics, however, in this case note that $p_0 = 0$ (since $b_1 = 0$) and hence β would not be used to update β_1. In all other cases, β would have the right semantics and by our inductive hypothesis, the update to β_1 is consistent. Now that we are done with the right sub-tree of the node, since the

left-most leaf of the left sub-tree of a node is the same as the left-most leaf of the sub-tree rooted at the same node, the update $\beta_0 = \beta$ is consistent. Furthermore, if that left-most leaf had index 1, we are not concerned with the semantics of β anyways. Thus, we have shown inductively that the β values are updated correctly during the downstream phase and are consistent with the semantics described above.

Output. We now finally show that $\mathcal{V}_{\text{pre}}$ is calculated correctly. Indeed, by our previous arguments,

$$\beta_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i-1)}^{i-1} v_j$$

as long as $i > 1$. Clearly, if $b_i = 0$, then $v'_i = v_i$ by definition and by our output procedure. If $b_i = 1$, note that $\mathsf{pre\text{-}ind}(i-1) = \mathsf{pre\text{-}ind}(i)$. Hence

$$v'_i = \beta_i \star v_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{i-1} v_j \star v_i = \bigstar_{j=\mathsf{pre\text{-}ind}(i)}^{i} v_j$$

as required.

The final algorithms we use can be found (implicitly) in the protocols described in Figure 6 and 7.

6 One to Many Join - $\Pi_{\mathsf{Join\text{-}OM}}$

We define the secret shared join functionality $\mathcal{F}_{\mathsf{Join\text{-}OM/MM}}$ in Figure 8. It takes as input the X, Y tables in secret shared form along with two user defined functions, $\mathsf{key}, \mathsf{Combine}$. The key function combines the join columns of both tables (domain $\mathbb{K}$). The $\mathsf{Combine}$ function is used to generate a row in the output table given a matching row from X and Y. The output table Z starts with a row corresponding to each row in Y. If some row Y_i does not have a matching row in X, then Z_i will be set to null. Otherwise Z_i is set to $\mathsf{Combine}(X_j, Y_i)$ where X_j is the unique row of X that matches Y_i, i.e. $\mathsf{key}(X_j) = \mathsf{key}(Y_i)$.

Our $\Pi_{\mathsf{Join\text{-}OM}}$ join protocol assumes table X w.l.o.g. has unique joining keys in the join column, while table Y w.l.o.g. has joining keys with an unbounded number of repeats in the join column. All other columns are unrestricted. The protocol is described in Figure 9 and requires four functionalities: secret shared sorting, permutations, aggregation and a generic MPC. When these are instantiated with our suggested protocols the overall running time is $O(n \log n)$ and requires $O(\log n)$ rounds of interaction. The protocol begins by sorting the join keys together into a single list. We use stable sort to ensure that in the case of matching keys, the key instance from table X will appear before the key instances from table Y. As a result, for every X_i with a matching rows $Y_{j_1}, ..., Y_{j_t}$, the X_i key is immediately followed by the set of keys $Y_{j_1}, ..., Y_{j_t}$. Conceptually, this simplifies the task of computing the join to obliviously appending row X_i onto the next t rows.

To improve efficiency, we further refine our method. Instead of actually sorting the combined keys, we generate a secret shared permutation $⟦\pi⟧$ that *if applied* to the combined $2n$ rows of X followed Y would stable sort them by their keys. We then actually apply the shared permutation $⟦\pi⟧$ to the table of combined keys K and a newly created table $X' := X//0^n$ which is the rows of X followed by n *appended null rows*. Let K'', X'' be the result respectively, i.e. $K'' = \pi(K), X'' = \pi(X')$. Note that for each i such that key K''_i was from Y, we have $X''_i = 0$ while for all other K''_i (which are from X) we have $X''_i = X_j$ for some $j \in [n]$.

PARAMETERS: Participating parties $\{P_0, \cdots, P_{m-1}\}$ Input vector $[\![\mathcal{V}]\!] \in \mathbb{D}^n$ and control bits $[\![\mathcal{B}]\!] \in \{0,1\}^n$. An aggregation type of $t \in \{\text{PREFIX, SUFFIX, FULL}\}$. A user defined circuit $\star : \mathbb{D}^2 \to \mathbb{D}$ that is associative.

PROTOCOL: Upon input $(t, \star, [\![\mathcal{V}]\!]_i, [\![\mathcal{B}]\!]_i)$ from party $P_i \in \{P_0, \cdots, P_{m-1}\}$, each party does the following:

1. **[Set Leaves]** If $t \in \{\text{PREFIX, FULL}\}$, assign $\rho_{i+n-1} := \mathcal{V}_i, \rho^*_{i+n-1} := \mathcal{B}_i$ for $i \in [n]$. If $t \in \{\text{SUFFIX, FULL}\}$, assign $\sigma_{i+n-1} := \mathcal{V}_i, \sigma^*_{i+n-1} := \mathcal{B}_{i+1}$ for $i \in [n]$.
2. **[Upstream]** For $d \in \{\log_2(n), ..., 0\}$, in parallel for $i \in [2^{d-1}, 2^d)$,
   (a) If $t \in \{\text{PREFIX, FULL}\}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, ([\![\rho_{2i}]\!], [\![\rho_{2i+1}]\!], [\![\rho^*_{2i}]\!], [\![\rho^*_{2i+1}]\!]))$ to obtain $([\![\rho_i]\!], [\![\rho^*_i]\!])$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\}^2 \to \mathbb{D} \times \{0,1\}$ is defined as
   $$\mathcal{C}(\rho_0, \rho_1, \rho^*_0, \rho^*_1) := \begin{cases} \rho_1, & \rho^*_1 = 0 \\ \rho_0 \star \rho_1, & \rho^*_1 = 1 \end{cases}, \rho^*_0\rho^*_1$$
   (b) If $t \in \{\text{SUFFIX, FULL}\}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, ([\![\sigma_{2i}]\!], [\![\sigma_{2i+1}]\!], [\![\sigma^*_{2i}]\!], [\![\sigma^*_{2i+1}]\!]))$ to obtain $([\![\sigma_i]\!], [\![\sigma^*_i]\!])$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\}^2 \to \mathbb{D} \times \{0,1\}$ is defined as
   $$\mathcal{C}(\sigma_0, \sigma_1, \sigma^*_0, \sigma^*_1) := \begin{cases} \sigma_0, & \sigma^*_0 = 0 \\ \sigma_0 \star \sigma_1, & \sigma^*_0 = 1 \end{cases}, \sigma^*_0\sigma^*_1$$
3. **[Downstream]** For $d \in \{0, ..., \log_2(n)\}$, in parallel for $i \in [2^{d-1}, 2^d)$,
   (a) If $t \in \{\text{PREFIX, FULL}\}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, ([\![\rho_i]\!], [\![\rho_{2i}]\!], [\![\rho^*_{2i}]\!]))$ to obtain $([\![\rho_{2i}]\!], [\![\rho_{2i+1}]\!])$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\} \to \mathbb{D}^2$ is defined as
   $$\mathcal{C}(\rho, \rho_0, \rho^*_0) := \rho, \begin{cases} \rho_0, & \rho^*_0 = 0 \\ \rho \star \rho_0, & \rho^*_0 = 1 \end{cases}$$
   (b) If $t \in \{\text{SUFFIX, FULL}\}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, ([\![\sigma_{2i+1}]\!], [\![\sigma_i]\!], [\![\sigma^*_{2i+1}]\!]))$ to obtain $([\![\sigma_{2i}]\!], [\![\sigma_{2i+1}]\!])$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\} \to \mathbb{D}^2$ is defined as
   $$\mathcal{C}(\sigma_1, \sigma, \sigma^*_1) := \begin{cases} \sigma_1, & \sigma^*_1 = 0 \\ \sigma_1 \star \sigma, & \sigma^*_1 = 1 \end{cases}, \sigma$$

Figure 6: Protocol Π_{Agg} for secret shared aggregation with no identity element (part 1 of 2).

4. **[Finalize]** Parallel for $i \in [n]$,

(a) If $t = \textsc{Prefix}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, [\![V_i]\!], [\![\rho_{i+n-1}]\!], [\![\rho^*_{i+n-1}]\!])$ to obtain $[\![V'_i]\!]$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\} \to \mathbb{D}$ is defined as

$$\mathcal{C}(v, \rho, \rho^*) := \begin{cases} v, & \rho^* = 0 \\ \rho \star v, & \rho^* = 1 \end{cases}$$

(b) If $t = \textsc{Suffix}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, [\![V_i]\!], [\![\sigma_{i+n-1}]\!], [\![\sigma^*_{i+n-1}]\!])$ to obtain $[\![V'_i]\!]$ where $\mathcal{C} : \mathbb{D}^2 \times \{0,1\} \to \mathbb{D}$ is defined as

$$\mathcal{C}(v, \sigma, \sigma^*) := \begin{cases} v, & \sigma^* = 0 \\ v \star \sigma, & \sigma^* = 1 \end{cases}$$

(c) If $t = \textsc{Full}$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, [\![V_i]\!], [\![\rho_{i+n-1}]\!], [\![\sigma_{i+n-1}]\!], [\![\rho^*_{i+n-1}]\!], [\![\sigma^*_{i+n-1}]\!])$ to obtain $[\![V'_i]\!]$ where $\mathcal{C} : \mathbb{D}^3 \times \{0,1\}^2 \to \mathbb{D}$ is defined as

$$\mathcal{C}(v, \rho, \sigma, \rho^*, \sigma^*) := \begin{cases} \mathcal{V}_i, & \rho^* = 0, \sigma^* = 0 \\ \mathcal{V}_i \star \sigma, & \rho^* = 0, \sigma^* = 1 \\ \rho \star \mathcal{V}_i, & \rho^* = 1, \sigma^* = 0 \\ \rho \star \mathcal{V}_i \star \sigma, & \rho^* = 1, \sigma^* = 1 \end{cases}$$

Figure 7: Protocol Π_{Agg} for secret shared aggregation with no identity element (part 2 of 2).

PARAMETERS: Participating parties $P_0, \cdots, P_{m-1}$. Database input tables $[\![X]\!], [\![Y]\!]$ with n rows. Let key be a local function that returns the join key column for $[\![X]\!], [\![Y]\!]$. Let $\mathsf{combine}$ be a circuit that computes an output row given matching rows from X, Y. Let pad be the function that determines how to pad the output table.

FUNCTIONALITY: Upon receiving $[\![X]\!], [\![Y]\!]$ from the parties,

1. Let $X := \mathsf{reconstruct}([\![X]\!]), Y := \mathsf{reconstruct}([\![Y]\!])$. Let $d = 0$.
2. For $i \in [n]$, and each $j \in [n]$ such that $\mathsf{key}(X_j) = \mathsf{key}(Y_i) \wedge \mathsf{isNull}(Y_i) = \textsc{False} \wedge \mathsf{isNull}(X_j) = \textsc{False}$, set $Z_d := \mathsf{Combine}(Y_i, X_j)$ and then $d = d + 1$.
3. Add dummy/null rows to Z until it has $D = \mathsf{pad}(d)$ rows.
4. Output $[\![Z]\!]_i$ to P_i where $[\![Z]\!] \leftarrow \mathsf{share}(Z)$.

Figure 8: Functionality $\mathcal{F}_{\mathsf{Join\text{-}OM/MM}}$ for secret shared Join for tables X and Y.

Importantly, for some row X''_i that originated in X with m matching rows in Y, X''_i will be followed by m null rows, with each corresponding to one of the matching rows from Y. As such, our goal will be to copy each such X''_i into the next m positions. In particular, for $i \in [n-1]$, if key $K''_i = K''_{i+1}$, then we want to copy X''_i into the next row X''_{i+1}. There are two interesting cases when analysing this step. If the current key K''_i is in the intersection and from X, then X''_i will not be null while X''_{i+1} will be null. The copy will overwrite the row X''_{i+1} with the desired result, i.e. $X''_{i+1} = X_j$ for some j. If K''_i is in the intersection and from Y then the same logic applies, we will copy X''_i (which was just copied in the previous iteration) into X''_{i+1}.

The second case is when the keys match but it is not in the intersection. Here, both of the keys must be from Y and corresponding rows in X'' must be null. As such, copying X''_i into X''_{i+1} will result in X''_{i+1} remaining null.

Instead of iteratively performing the copies of X''_i into X''_{i+1}, which would take $O(n)$ rounds, we will use an aggregation tree, which will only require $O(\log n)$ rounds. We apply our prefix aggregation by setting the control bit condition $\mathcal{B}_i$ to 1 iff $K''_{i-1} = K''_i$, using X'' as our input vector and defining our aggregation operator as $x_0 \star x_1 := x_0$. The complexity of prefix aggregation is $O(n \log n)$ time and requires $O(\log n)$ rounds. The final task is to map the copied X'' rows back to the Y table. By applying the inverse permutation of $(\![\pi]\!)$ to X'' we get table X^*. The first n rows of X^* will still contain the same value as the first n rows of X''. However, the last n rows of X^* are now either the rows from X what matched with Y or null. We can easily associate the ith row of Y and $(n+i)$-th row of X^* to get the ith output row.

To instantiate our protocol we observe that secret-shared sorting in the three party honest majority setting is an essential building block. Further, our join protocols have a nice feature that any new efficient construction for sorting can easily be plugged in for improved performance. In this work, the employ the state-of-the-art sorting protocol proposed by Chida et al. [CHI+19b].

Theorem 6.1. *If m computing parties that hold secret shares of two input database tables $[\![X]\!], [\![Y]\!]$, each with n rows and unique join keys in table X. Let $\mathsf{pad}(d) = n$. In the semi-honest $\mathcal{F}_{\mathsf{Sort}}$, $\mathcal{F}_{\mathsf{Perm}}$, $\mathcal{F}_{\mathsf{Agg}}$, $\mathcal{F}_{\mathsf{MPC}}$ hybrid model, protocol Figure 9 securely realizes functionality Figure 8.*

Proof. The protocol in Figure 9 securely realizes Figure 8 in the $\mathcal{F}_{\mathsf{Sort}}$, $\mathcal{F}_{\mathsf{Perm}}$, $\mathcal{F}_{\mathsf{Agg}}$, $\mathcal{F}_{\mathsf{MPC}}$ hybrid

PARAMETERS: Participating parties $P_0, \cdots, P_{m-1}$. Database input tables $[\![X]\!], [\![Y]\!]$ with n rows where X is required to have unique join keys. Let key be a local function that returns the join key column for $[\![X]\!], [\![Y]\!]$. Let $\mathsf{combine}$ be a circuit that computes an output row given matching rows from X, Y.

PROTOCOL: Each party $P_i \in \{P_0, P_1, \cdots, P_{m-1}\}$ does the following:

1. **[Get keys]** The parties locally construct the list $[\![K]\!] \in \mathbb{K}^{2n}$ as the concatenation of the join keys $\mathsf{key}([\![X]\!])$ followed by $\mathsf{key}([\![Y]\!])$.
2. **[Sort keys]** The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on $(\textsc{Sort}, [\![K]\!])$ and obtain the secret shared permutation $(\![\pi]\!)$.
3. **[Dummy rows]** Locally, the parties prepend n `Null` rows to $[\![X]\!]$ to obtain $[\![X']\!]$.
4. **[Permute X]** The parties invoke $\mathcal{F}_{\mathsf{Perm}}$ on $(\textsc{Perm}, (\![\pi]\!), [\![X']\!]), (\textsc{Perm}, (\![\pi]\!), [\![K]\!])$ to obtain $[\![X'']\!], [\![K'']\!]$, respectively.
5. **[Control bits]** For $i \in [2n-1]$ the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, ([\![K''_i]\!], [\![K''_{i+1}]\!]))$ to obtain $[\![\beta_{i+1}]\!]$ where the circuit $\mathcal{C} : \mathbb{K}^2 \to \{0,1\}$ outputs 1 iff the two $\mathbb{K}$ elements are equal. The parties locally define $[\![\beta_1]\!] := [\![0]\!]$.
6. **[Duplicate]** The parties invoke $\mathcal{F}_{\mathsf{Agg}}$ on $(\textsc{Prefix}, \mathsf{dup}, [\![X'']\!], [\![\beta]\!])$ to obtain $[\![X''']\!]$ where $\mathsf{dup}(x_0, x_1) := x_0$.
7. **[Unpermute X]** The parties invoke $\mathcal{F}_{\mathsf{Perm}}$ on $(\textsc{InvPerm}, (\![\pi]\!), [\![X''']\!])$ to obtain $[\![X^*]\!]$.
8. **[Combine]** Parallel for $i \in [n]$, the parties invoke $\mathcal{F}_{\mathsf{MPC}}$ on $(\mathcal{C}, [\![Y_i]\!], [\![X^*_i]\!])$ and obtain $[\![Z_i]\!]$ where $\mathcal{C}$ is defined as

$$\mathcal{C}(x,y) := \begin{cases} (\mathsf{Combine}(x,y), \textsc{False}), & \neg\mathsf{isNull}(x) \wedge \neg\mathsf{isNull}(y) \\ (0, \textsc{True}), & \mathsf{isNull}(x) \vee \mathsf{isNull}(y) \end{cases}$$

The parties output $[\![Z]\!]$.

Figure 9: Protocol $\Pi_{\mathsf{Join\text{-}OM}}$ for secret shared Join for one table X with unique join keys.

model. Each party P_i's view consists of their input shares $[\![X]\!]_i, [\![Y]\!]_i$ (according to a scheme secure in the honest/dishonest semi-honest setting) and output received from the ideal functionalities $\mathcal{F}_{\mathsf{Sort}}$, $\mathcal{F}_{\mathsf{Perm}}$, $\mathcal{F}_{\mathsf{Agg}}$, $\mathcal{F}_{\mathsf{MPC}}$. By the simulatability of the secret sharing scheme, the shared returned to the corrupt parties from these functionalities can be simulated without the output. As such, the simulator will do exactly this, instead of forwarding the inputs to the ideal functionalities, the simulator will instead simply sample uniformly random output shares for each functionality and return these. This change is identically distributed. For the final shares generated in step 8, the simulator returns the shares output by $\mathcal{F}_{\mathsf{Join\text{-}OM/MM}}$ instead of calling $\mathcal{F}_{\mathsf{MPC}}$. Correctness of the protocol can be verified by inspection. This completes the simulation.

Remark. Our protocol can be extended to work in the malicious setting. We can write a simulator for the malicious setting given the ideal functionalities. However, given some subtleties about making generic statements in the malicious setting (input extraction) we defer formal proofs to future work.

7 Many to Many Join - $\Pi_{\mathsf{Join\text{-}MM}}$

We present our many to many join protocol in Figure 11 & 12. As an overview, we first sort the tables together by the join key. For each row we count the multiplicity of the key in table X and Y. This is done using an aggregation tree. With this information we can compute how many copies of each row are needed. We then re-order/sort the rows so that each row has the d dummy rows after it where d is the number of copies that are required. We then copy each such row into the next d rows. Finally, for table Y we re-order/sort it again to match the ordering of X.

For each of these reorderings, only the columns/attributes that are required are reordered. In the end we permute the actual X, Y tables by a “composed reordering.” This allows the protocol to permute less data.

The protocol proceeds in a few steps starting with sorting the combined set of keys from both X and Y as done in Step 2. The keys K themselves are sorted along with flag vectors t, N indicating which table each rows is from and whether that row is null.

Next is Step 3 where the number of times (multiplicity) each key appears in X and Y is computed. We will store these counts in the matrix $M \in \mathbb{Z}^{2n\times 2}$ where the first column stores the number of times the current key appears in X and the second column Y.

During this step we compute two sets of control bits $\mathcal{B}, \mathcal{B}'$. The first defines blocks where each block contains only equal keys. These will be used for this step. The second further refines this to have equal keys and *be from the same table*. We will make use of $\mathcal{B}'$ later on.

For rows from X, the corresponding row in M is initialized as (1,0) while this is reversed for rows from Y. We can then perform a full aggregation with addition on control bits $\mathcal{B}$. This results in each row of M counting the multiplicity of the key for table X and separately for table Y.

Step 4 then splits the tables back apart except that they are now in sorted order. This is done by sorting by t. In particular, the first n rows of $M, \mathcal{B}', N$ correspond to table X while the remaining correspond to Y. Note, we could also sort the tables by similarly applying the permutation $\pi' \circ \pi$ to the combined table $X//Y$ but don’t to reduce the amount of data being permuted.

Step 5 computes the index/position in the output table that the key corresponding to the current row will first appear at. This computation is only needed for the rows in the Y table. In more detail, each row i knows that it’s key appears $M_{i,1}$ times in X and $M_{i,2}$ times in Y and therefore will appear $p_i := M_{i,1} \cdot M_{i,2}$ times in the output table. The idea is to then compute

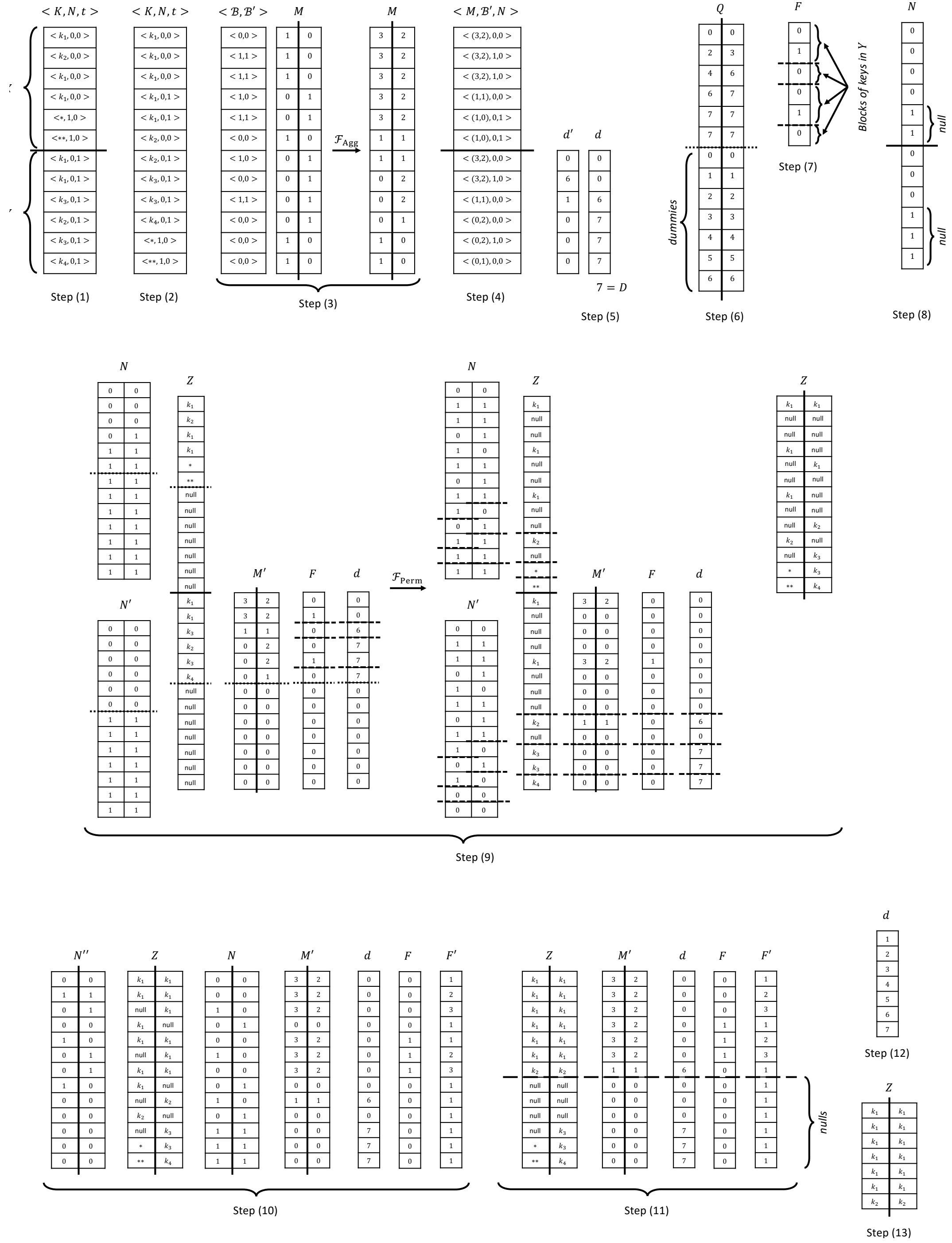

Figure 10: Illustration of protocol $\Pi_{\mathsf{Join\text{-}MM}}$. Set X has records with keys k_1, k_2, k_1, k_1 and set Y has records with keys $k_1, k_1, k_3, k_2, k_3, k_4$, where $k_1 < k_2 < k_3 < k_4$. Note that $n = 6$ and X will be padded with two null rows.

$d'_i := p_i \cdot c_i$ where c_i is a condition bit that is one iff this row is the last in the block of equal keys, i.e., $c_i := \neg\mathcal{B}'_{i+1}$ is the inverse of the next control bit which is zero iff row $i+1$ is the start of a new block.

The result is that d'_i for the last row of a block contains how much space that block requires. Each row i can then compute the starting position of their block in the output table as $d_i := \sum_{j\in[i)} d'_j$ which for $i > 1$ is equal to $d_{i-1} + d'_{i-1}$. Later we will update these d_i values to be the exact row that each Y row should be mapped to as opposed to the start of the block.

Additionally, in this step we compute the size of the output table as $D := d_{n+1}$. If desired, a possibly randomized upper bound of this value can also be computed with the maximum possible being n^2. However, our default is to reveal the exact value.

Step 6 performs a similar computation as Step 5 except we compute the index that this row should be at such that there are m dummy rows after it where m is the multiplicity of this row in the other table, i.e. $m := M_{i,2}$ for X rows and $m := M_{i,1}$ for Y rows. We then compute the overall index Q_i of this row as the sum of all previous m values. Finally, we append D null rows to each table where the Q index of these rows are $1, 2, ..., D$.

The idea is then to sort the tables by their Q index. Row i will then be followed by dummy rows with Q indices $Q_i, Q_i + 1, ..., Q_i + m - 1$.

Sorting is performed Step 9 but before that, in Step 8 the null flags for each row are updated so that unmatched rows are marked as null. These rows will removed later.

Step 10 will then duplicate the rows into the following rows as required. In particular, if a row needs m duplicates, it will be followed by m dummy rows. However, if we copied the row into all of these there would be $m+1$ copies of it. As such we copy the row $m-1$ times. This is controlled by the N'' control bits. Step 11 then removes the excess dummy rows. It is helpful to think of N as the indicator for the null rows, N' as the indicator for the dummy rows, and N'' as the indicator for the excess dummy rows that are to be treated as null and removed.

Finally, in Step 12 we compute the local indices that each Y row should be mapped to. Recall that we previously computed the location d_i in the output where the key K_i will start appearing. We now need to compute the *offset* from this location that the current row will reside at. During Step 3 we computed the multiplicity of the key in the X table, $M_{*,1}$. In Step 7 we computed which offset indicies that tell each row of Y that it is the F_i'th row with that key. Finally, in Step 10, we additionally compute which duplicate that each row is. Putting this all together, we can compute where each Y row should be mapped to as

$$d_i + F_i M_{i,1} + F'_i$$

It is then a simple matter of sorting the Y rows by these indices and outputting the concatenation of the X rows with the result.

Modeling D as Leakage. The many-to-many protocol reveals the output table size D to all parties. As discussed in the introduction, D does not need to be the true output table size, as done in $\Pi_{\mathsf{Join\text{-}MM}}$, step 5. Revealing D exactly has the advantage of being the most computationally efficient method but also leaks some information about the input tables. Depending on application this leakage could be non-trivial and be considered a privacy violation. However, in most of the our motivating application this does not lead to a significant harm to privacy. On the other hand, revealing no information at all is possible by always setting $D = n^2$, the largest value possible. In this case the protocol is designed to add the necessary number dummy rows such that the output

Parameters: Participating parties $P_0, \cdots, P_{m-1}$. Database input tables $[X], [Y]$ with n rows and possibly duplicate join keys. Let key be a local function that returns the join key column for $[X], [Y]$. Let combine be a circuit that computes an output row given matching rows from X, Y.

Protocol: Each party does the following:

1. **[Extract]** The parties locally construct the lists

$$[K] := \mathsf{key}([X]) /\!/ \mathsf{key}([Y])$$
$$[N] := \mathsf{isNull}([X]) /\!/ \mathsf{isNull}([Y])$$
$$[t] := [0^{n\times 1}] /\!/ [1^{n\times 1}]$$

2. **[Sort by key]** The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on $(\textsc{Sort}, [K])$ and obtain the secret shared permutation $⟦\pi⟧$. They invoke $\mathcal{F}_{\mathsf{Perm}}$ on

$$(\textsc{Perm}, ⟦\pi⟧, ([K], [t], [N]))$$

and store the result as $([K], [t], [N])$.

3. **[Multiplicity]** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute $[\mathcal{B}], [\mathcal{B}'] \in \{0,1\}^{2n}$ as $\mathcal{B}_1 := 0, \mathcal{B}'_1 := 0$ and

$$[\mathcal{B}_i] := ([K_i] = [K_{i-1}]) \wedge \neg[N_i],$$
$$[\mathcal{B}'_i] := (\neg[t_i] \oplus [t_{i-1}]) \wedge [\mathcal{B}_i]$$

The parties invoke $\mathcal{F}_{\mathsf{Convt}}$ on $(\mathbb{Z}, [t])$ to obtain $\langle\!\langle t \rangle\!\rangle$ and then store $\langle\!\langle M \rangle\!\rangle := (1^{2n\times 1} - \langle\!\langle t \rangle\!\rangle) \| \langle\!\langle t \rangle\!\rangle \in \mathbb{Z}^{2n\times 2}$.

The parties invoke $\mathcal{F}_{\mathsf{Agg}}$ on inputs

$$(\textsc{Full}, +, \langle\!\langle M \rangle\!\rangle, [\mathcal{B}])$$

and store the results as $\langle\!\langle M \rangle\!\rangle$.

4. **[Partition tables]** The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on input $[t]$ and obtains $⟦\pi'⟧$ as the result. The parties invoke $\mathcal{F}_{\mathsf{Perm}}$ on inputs

$$(\textsc{Perm}, ⟦\pi'⟧, (\langle\!\langle M \rangle\!\rangle, [\mathcal{B}'], [N]))$$

and store the results as $(\langle\!\langle M \rangle\!\rangle, [\mathcal{B}'], [N])$.

5. **[Destination index]** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute

$$\langle\!\langle d' \rangle\!\rangle := \langle\!\langle M_{(n,2n),1} \rangle\!\rangle \cdot \langle\!\langle M_{(n,2n),2} \rangle\!\rangle \cdot \neg[\mathcal{B}'_{(n+1,2n]}]$$
$$\langle\!\langle d'_n \rangle\!\rangle := \langle\!\langle M_{n,1} \rangle\!\rangle \cdot \langle\!\langle M_{(n,2} \rangle\!\rangle,$$
$$\langle\!\langle d_1 \rangle\!\rangle := 0,$$
$$\langle\!\langle d_{i+1} \rangle\!\rangle := \langle\!\langle d_i \rangle\!\rangle + \langle\!\langle d'_i \rangle\!\rangle \quad \text{for } i \in [n]$$

The parties reveal $D := \mathsf{reconstruct}(\mathsf{pad}(\langle\!\langle d_{n+1} \rangle\!\rangle))$ and update $\langle\!\langle d \rangle\!\rangle = \langle\!\langle d_{[n]} \rangle\!\rangle$.

6. **[Dummy index]** The parties locally compute $\langle\!\langle Q \rangle\!\rangle \in \mathbb{Z}^{(n+D)\times 2}$ as $Q_{1,1} := 0, Q_{1,2} := 0$ and for $i \in [n)$,

$$\langle\!\langle Q_{i+1,1} \rangle\!\rangle := \langle\!\langle M_{i,2} \rangle\!\rangle + \langle\!\langle Q_{i,1} \rangle\!\rangle$$
$$\langle\!\langle Q_{i+1,2} \rangle\!\rangle := \langle\!\langle M_{i+n,1} \rangle\!\rangle + \langle\!\langle Q_{i,2} \rangle\!\rangle$$

and $Q_{i+n-1,1} := i, Q_{i+n-1,2} := i-1$ for $i \in [D]$.

Figure 11: Protocol $\Pi_{\mathsf{Join\text{-}MM}}$ for secret shared many to many Join where both tables can have duplicate join keys (part 1 of 2).

7. **[Offset index]** The parties invoke $\mathcal{F}_{\mathsf{Agg}}$ on
$$(\mathsf{Prefix}, +, \langle\!\langle 1^{n\times 1}\rangle\!\rangle, [\mathcal{B}'_{(n,2n]}])$$
and stores the result as $\langle\!\langle F\rangle\!\rangle$. Set $\langle\!\langle F\rangle\!\rangle := \langle\!\langle F\rangle\!\rangle - 1^{n\times 1}$.

8. **[Null flag]** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute
$$[N_{[n]}] := [N_{[n]}] \vee (\langle\!\langle M_{[n],2}\rangle\!\rangle = 0^n)$$
$$[N_{(n,2n]}] := [N_{(n,2n]}] \vee (\langle\!\langle M_{[n],1}\rangle\!\rangle = 0^n)$$

9. **[Sort dummies]** The parties set
$$[Z] := [X] /\!/ [\texttt{null}^D] /\!/ [Y] /\!/ [\texttt{null}^D]$$
$$[N] := ([N_{[n]}] /\!/ [1^{D\times 1}]) \| ([N_{(n,2n]}] /\!/ [1^{D\times 1}])$$
$$[N'] := ([0^{n\times 1}] /\!/ [1^{D\times 1}]) \| ([0^{n\times 1}] /\!/ [1^{D\times 1}])$$
$$[M'] := \langle\!\langle M_{(n,2n]}\rangle\!\rangle /\!/ \langle\!\langle 0^{D\times 2}\rangle\!\rangle$$
$$[F] := \langle\!\langle F\rangle\!\rangle /\!/ \langle\!\langle 0^{D\times 1}\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle := \langle\!\langle d\rangle\!\rangle /\!/ \langle\!\langle 0^{D\times 1}\rangle\!\rangle$$

The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on $\langle\!\langle Q_{*,1}\rangle\!\rangle, \langle\!\langle Q_{*,2}\rangle\!\rangle$ and receive $(\!|\pi^{Q,1}|\!), (\!|\pi^{Q,2}|\!)$. The parties define $(\!|\pi^*|\!) := ((\!|\pi^{Q,1}|\!) \| (\!|\pi^{Q,2}|\!)) \circ (((\!|\pi'|\!) \circ (\!|\pi|\!))_{[n]} \| \mathcal{I}_D \| ((\!|\pi'|\!) \circ (\!|\pi|\!))_{(n,2n]} \| \mathcal{I}_D)$ and invoke $\mathcal{F}_{\mathsf{Perm}}$ on
$$(\mathsf{Perm}, (\!|\pi^*|\!), [Z]),$$
$$(\mathsf{Perm}, (\!|\pi^{Q,1}|\!), ([N_{*,1}], [N'_{*,1}])),$$
$$(\mathsf{Perm}, (\!|\pi^{Q,2}|\!), ([N_{*,2}], [N'_{*,2}], \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle, [M']))$$
and store the results as $[Z], ([N_{*,1}], [N'_{*,1}]), ([N_{*,2}], [N'_{*,2}], \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle, \langle\!\langle M'\rangle\!\rangle)$. Set $[Z] := [Z_{[n+D]}] \| [Z_{(n+D,2n+2D]}]$.

10. **[Duplicate]** Invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute $[N''] := [N'_{[n+D)}] \cdot [N'_{(n+D]}] /\!/ [0^{1\times 2}]$. Invoke $\mathcal{F}_{\mathsf{Agg}}$ on
$$(\textsc{Prefix}, \mathsf{dup}, ([Z_{*,1}], [N_{*,1}]), [N''_{*,1}]),$$
$$(\textsc{Prefix}, \mathsf{dup}, ([Z_{*,2}], [N_{*,2}], \langle\!\langle M'\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle), [N''_{*,2}]),$$
$$(\textsc{Prefix}, +, \langle\!\langle 1^{(n+D)\times 1}\rangle\!\rangle, [N''_{*,2}])$$
where $\mathsf{dup}(x_0, x_1) = x_0$ and store the result as $([Z_{*,1}], [N_{*,1}]), ([Z_{*,2}], [N_{*,2}], \langle\!\langle M'\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle), \langle\!\langle F'\rangle\!\rangle$.

11. **[Partition nulls]** The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on inputs $[N_{*,1}], [N_{*,2}]$ and store the results as $(\!|\pi^{N,1}|\!), (\!|\pi^{N,2}|\!)$. The parties invoke $\mathcal{F}_{\mathsf{Perm}}$ on inputs
$$(\mathsf{Perm}, (\!|\pi^{N,1}|\!), [Z_{*,1}])$$
$$(\mathsf{Perm}, (\!|\pi^{N,2}|\!), ([Z_{*,2}], \langle\!\langle M'\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle, \langle\!\langle F'\rangle\!\rangle))$$
and store the results as $[Z_{*,1}], ([Z_{*,2}], \langle\!\langle M'\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle, \langle\!\langle F'\rangle\!\rangle)$. The parties drop the last n rows by setting $[Z] := [Z_{[D]}], \langle\!\langle M'\rangle\!\rangle := \langle\!\langle M'_{[D]}\rangle\!\rangle, \langle\!\langle d\rangle\!\rangle := \langle\!\langle d_{[D]}\rangle\!\rangle, \langle\!\langle F\rangle\!\rangle := \langle\!\langle F_{[D]}\rangle\!\rangle, \langle\!\langle F'\rangle\!\rangle := \langle\!\langle F'_{[D]}\rangle\!\rangle$.

12. **[Destination index]** The parties invoke $\mathcal{F}_{\mathsf{MPC}}$ to compute $\langle\!\langle d\rangle\!\rangle := \langle\!\langle d\rangle\!\rangle + \langle\!\langle F\rangle\!\rangle \cdot \langle\!\langle M'_{*,1}\rangle\!\rangle + \langle\!\langle F'\rangle\!\rangle$.

13. **[Sort by destination]** The parties invoke $\mathcal{F}_{\mathsf{Sort}}$ on $\langle\!\langle d\rangle\!\rangle$ and stores the result as $(\!|\pi^d|\!)$. The parties invoke $\mathcal{F}_{\mathsf{Perm}}$ on $(\mathsf{Perm}, (\!|\pi^d|\!), [Z_{*,2}])$ and store the result as $[Z_{*,2}]$. The parties output $[Z]$.

Figure 12: Protocol $\Pi_{\mathsf{Join\text{-}MM}}$ for secret shared many to many Join where both tables can have duplicate join keys (part 2 of 2).

		LAN Time (sec.)				Total Communication (MB)			
		n				n			
Operation	Protocol	2^8	2^{12}	2^{16}	2^{20}	2^8	2^{12}	2^{16}	2^{20}
One-to-One Joins	[MRR20]	0.02	0.05	0.5	12.3	0.3	4.9	78	1,249
	[LTW13a]*	2.0	8.0	128.0	*2,048.0	–	–	–	–
One-to-Many Joins	This	0.09	0.21	1.3	21.6	1.5	22.8	364	5,560
Many-to-Many Joins	This	0.27	0.81	7.3	129.2	8.0	129.5	2,110	32,910

Figure 13: The running time in seconds and communication overhead in MB for various join operations and application. The input and output tables are of size n. * denotes that the running times were linearly extrapolated from the values of n provided by the publication.

table has D rows. Unfortunately this approach is impractical when n is sufficiently large, e.g. $n = 2^{20}$. Therefore some information leakage is unavoidable if the protocol is to be practical.

To reduce leaking a little, one could round D up to the next power of two. This would not significantly impact performance while limiting the amount of information revealed. Another [complementary] approach would be to use differential privacy, where a random noisy upper bound on D is revealed [GMRW13]. We leave the optimal way of choosing D to future work and simply suggest that D is either revealed exactly or rounded up to a power of 2, depending on application specific considerations.

Theorem 7.1. *Given m computing parties that hold secret shares of two input database tables $[\![X]\!], [\![Y]\!]$, each with n rows. Let $\mathsf{pad}(d) = d$. In the semi-honest $\mathcal{F}_{\mathsf{Sort}}$, $\mathcal{F}_{\mathsf{Perm}}$, $\mathcal{F}_{\mathsf{Agg}}$, $\mathcal{F}_{\mathsf{MPC}}$ hybrid model, protocol Figure 11, Figure 12 securely realizes Figure 8.*

Proof. Similar to proof of Theorem 6.1. □

8 Extensions

We note that a variety of extensions can be applied to our core protocols. For example, our core protocols are designed to compute inner joins between the two tables. However, it is a relatively simple task to extend our techniques to left, right and full joins along with unions. We define the different kinds of joins in appendix Appendix A. Additionally, it is a simple extension to apply some additional computation to the output table. For example, a `where` clause in the output table can filter the results as required. We refer the reader to [MRR20] for a detailed description of these extensions.

9 Evaluation

We implement our protocols in the honest majority three party framework of [MR18a]. All experiments for us and [MRR20] were performed on a consumer grade laptop with a Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz processor and 16GB of RAM. Networking was performed using TCP via localhost with sub millisecond latency. A single thread per party was used. All cryptographic operations are performed with computational security parameter $\kappa = 128$ and statistical security $\lambda = 40$. We consider set/table sizes of $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$. For $\Pi_{\mathsf{Join\text{-}MM}}$ we set $D = n$. In

general $\Pi_{\textsf{Join-MM}}$ scales with the maximum of n, D. The input join columns sizes are 32 bits for our protocols.

In Figure 13 we report our performance numbers and comparison to two other protocols [MRR20, LTW13a]. The first is [MRR20] which aims to offer a similar functionality to ours with the major restriction that the input tables to their protocol must have *unique* keys while our protocols allow duplicates in one or both tables.

The main advantage of their approach is that their running time is linear in the inputs sets while ours is $O(n \log n)$. However, this difference is somewhat deceiving due to their protocol having linear dependency on the security parameter, i.e., a running time of $O(n\kappa)$, due to the need to evaluate the LowMC block cipher [ARS+15] within the MPC. Regardless, their protocol is mildly faster, requiring 12.3 seconds to join two tables of a million items compared to 21.6 seconds for our One-to-Many protocol $\Pi_{\textsf{Join-OM}}$. We argue that such a minor increase in running time is well worth the added functionality when it is required. Similarly, their protocol requires less communication, with a total of $1.2GB$ of data sent compared to $\Pi_{\textsf{Join-OM}}$ sending $5.6GB$ when joining tables of a million items.

We also compare to [LTW13a] which has the same restriction as [MRR20] but is overall much slower and requires more communication. We were not able to obtain concrete communication numbers for them and the running times are taken from their publication.

We also observe that our $\Pi_{\textsf{Join-OM}}$ protocol is approximately 6 times faster than our more general $\Pi_{\textsf{Join-MM}}$ protocol. In particular, this is due to the added support for allowing duplicates in both tables as opposed to just one of them. In more detail, $\Pi_{\textsf{Join-OM}}$ sorts the two tables together just once and then can perform efficient, linear time operations for the remaining of the protocol. Unfortunately, $\Pi_{\textsf{Join-MM}}$ requires sorting several times. The first is sorting the input join columns which scales linearly in $O(n\ell) = O(n \log n)$ where ℓ is the bit length of the input join column. Both $\Pi_{\textsf{Join-OM}}$ and $\Pi_{\textsf{Join-MM}}$ require this computation. However, after that, $\Pi_{\textsf{Join-MM}}$ requires two additional sorts with overhead $O((n + D) \log D)$ where D is the upper bound on the output table size. In particular, our Step 9 must sort two lists of length $n + D$ with keys of size $\log(D)$. This is the most expensive operation in the protocol and takes up about 80% of the running time. The last sort operation is to rearrange the order of the duplicated Y table which requires $O(D \log D)$ overhead.

Due to the need to sort the join column and this being the main overhead for $\Pi_{\textsf{Join-OM}}$, this protocol scales almost linearly in the column bit length. For example, increasing it from 32 to 64 in our experiments should roughly double the overhead. However, this is not strictly the case for $\Pi_{\textsf{Join-MM}}$ where the main overhead is sorting in step Step 9 which scales as $O((n + D) \log D)$. Moreover, if the key length gets too long then they can be compressed using a randomized encoding technique as described in [MRR20]. This effectively ensures that the bit length of the join column can never be larger than $\lambda + 2 \log n$.

References

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia,*

Bulgaria, April 26-30, 2015, Proceedings, Part I, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[BA11] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. Cryptology ePrint Archive, Report 2011/464, 2011. https://ia.cr/2011/464.

[CHI+19a] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. *IACR Cryptol. ePrint Arch.*, page 695, 2019.

[CHI+19b] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive, Report 2019/695, 2019. https://ia.cr/2019/695.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled psi from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, Canada, October 14 - 16, 2018*. ACM, 2018.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, 2017.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. Cryptology ePrint Archive, Report 2018/105, 2018. https://ia.cr/2018/105.

[CTB+18] Rémi Canillas, Rania Talbi, Sara Bouchenak, Omar Hasan, Lionel Brunie, and Laurent Sarrat. Exploratory study of privacy preserving fraud detection. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 25–31, New York, NY, USA, 2018. Association for Computing Machinery.

[DPDea20] Morten Dahl, Justin Patriquin, Yann Dupis, and et. al. Tf encrypted: Encrypted deep learning in tensorflow. 2020.

[Fac20] Facebook. Crypten: A research tool for secure machine learning in pytorch. 2020.

[GMRW13] S. Dov Gordon, Tal Malkin, Mike Rosulek, and Hoeteck Wee. Multi-party computation of polynomials and branching programs without simultaneous interaction. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 575–591. Springer, 2013.

[GPR+21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 395–425. Springer, 2021.

[IKN+17a] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[IKN+17b] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *IACR Cryptology ePrint Archive*, 2017:738, 2017.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *CCS*, 2016.

[KLS+17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proc. Priv. Enhancing Technol.*, 2017(4):177–197, 2017.

[KMP+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS*, 2017.

[Lin16] Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Paper 2016/046, 2016. https://eprint.iacr.org/2016/046.

[LKFV21] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. Secrecy: Secure collaborative analytics on secret-shared data. *CoRR*, abs/2102.01048, 2021.

[LTW13a] Sven Laur, Riivo Talviste, and Jan Willemson. From oblivious aes to efficient and secure database join in the multiparty setting. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 84–101, Berlin, Heidelberg, 2013. Springer-Verlag.

[LTW13b] Sven Laur, Riivo Talviste, and Jan Willemson. From oblivious aes to efficient and secure database join in the multiparty setting. In *International Conference on Applied Cryptography and Network Security*, pages 84–101. Springer, 2013.

[Met22] Meta. What are privacy-enhancing technologies (pets) and how will they apply to ads?, 2022. https://about.fb.com/news/2021/08/privacy-enhancing-technologies-and-ads/.

[MR18a] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. *IACR Cryptology ePrint Archive*, 2018:403, 2018.

[MR18b] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. https://ia.cr/2018/403.

[MRR20] Payman Mohassel, Peter Rindal, and Mike Rosulek. *Fast Database Joins and PSI for Secret Shared Data*, page 1271–1287. Association for Computing Machinery, New York, NY, USA, 2020.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n ot extension with application to private set intersection. In Helena Handschuh, editor, *Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*, pages 381–396, Cham, 2017. Springer International Publishing.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO*, 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In *EUROCRYPT*, 2020.

[PSSZ15a] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530. USENIX Association, 2015.

[PSSZ15b] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX*, 2015.

[PSTY19a] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT*, 2019.

[PSTY19b] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based psi with linear communication. Cryptology ePrint Archive, Report 2019/241, 2019. https://ia.cr/2019/241.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT*, 2018.

[PSZ14a] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, San Diego, CA, 2014. USENIX Association.

[PSZ14b] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *USENIX*, 2014.

[PSZ16] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930, 2016. http://eprint.iacr.org/2016/930.

[RA18] Amanda Cristina Davi Resende and Diego F. Aranha. Faster unbalanced private set intersection. 2018.

[RR22] Peter Rindal and Srinivasan Raghuraman. Blazing fast psi from improved okvs and subfield vole. Cryptology ePrint Archive, Report 2022/320, 2022. https://ia.cr/2022/320.

[RS21] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in*

Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II, volume 12697 of *Lecture Notes in Computer Science*, pages 901–930. Springer, 2021.

[RSC+19] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: xnor-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1501–1518. USENIX Association, 2019.

[SvHA+19] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty pagerank algorithm for collaborative fraud detection. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 605–623, Cham, 2019. Springer International Publishing.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, 2019.

[Wil19] John Wilander. Privacy preserving ad click attribution for the web., 2019. `webkit.org/blog/8943/privacy-preserving-ad-click-attribution-for-the-web/`.

A SQL-like Join Operations

We can view a database X as a collection of rows $X[i]$ with $i \in \{0, \cdots, n\}$. A database join on input databases X, Y is defined by identifying a join column X^k, Y^k in both tables. Next, we match values in both columns X^k, Y^k to determine the rows of the output table. Below we list the constraints for different types of joins -

1. Inner Join: Every pair (x_i, y_j) of matching values where $x_i = y_j$ in the join columns (X^k, Y^k) corresponds to the a row in the output table. Each row in the output table is a concatenation of the rows $(X[i] \parallel Y[j])$ corresponding to the matching keys from both tables.
2. Left/Right Join: If we assign X as left table and Y as right table, then left join is defined as X inner join Y plus all the remaining rows of the left table X (all the rows of X that are not in the inner join). A right join is defined symmetrically as X inner join Y along with all the remaining rows of the right table Y (without the matching values in the join column). All the columns with missing values are assigned NULL.
3. Union: We write $X \cup Y = X$ left join $(Y \backslash X)$. The output table consists of all the rows of the left table X and all the rows of table Y that are not in the inner join (all the rows without a matching value in table X) with NULL in all the missing columns.
4. Full Join: This operation can be expressed as $\{X \cap Y\} \cup \{Y \backslash X\} \cup \{X \backslash Y\}$ and can be written as (X Left join Y) union Y or (Y left join X) union X.

Beyond joins, this framework allows for selecting rows in a table that meet a certain criterion or computing the aggregation values (sum , count, max value) over a column in a single database In this paper, we describe the protocol to obliviously compute the inner join. We refer the reader to a previous work [MRR20] that describes the generic extension from inner join to the full suite of join operations, aggregation and filtering using the where clause.

B Aggregation trees

B.1 Suffix Aggregation

Theorem B.1. *Given input array $\mathcal{V} = [v_1, v_2, \ldots, v_n]$ with associated control bits $\mathcal{B} = [b_1 = 0, b_2, \ldots, b_n]$, protocol Figure 4 correctly computes the suffix aggregation $\mathcal{V}' = [v'_1, v'_2, \ldots, v'_n]$ where $v'_i = \bigstar_{j=i}^{\textsf{suf-ind}(i)} v_j$.*

Proof. To see that the above algorithm indeed performs suffix aggregation, we set up the following semantics for each of the entries in the 5-tuple. For an internal node with an associated 5-tuple $(\gamma, \delta, \textsf{suffix}, \ell, p)$,

- δ is the sum of the leaves of the sub-tree rooted at the internal node, starting from the left until and excluding the first value whose corresponding control bit is 0 (if it exists). More formally,
$$\delta = \bigstar_{j=i_{\text{left}}}^{\min\{\textsf{suf-ind}(i_{\text{left}}), i_{\text{right}}\}} v_j$$
where i_{left} and i_{right} are the indices of the left-most and right-most leaves of the sub-tree rooted at the internal node respectively.

- γ is the δ value of the right sub-tree of the sub-tree rooted at the internal node.
- suffix is the suffix aggregation value of the right-most leaf of the sub-tree rooted at the internal node, excluding the value of the right-most leaf. More formally,

$$\mathsf{suffix} = \bigstar_{j=i_{\text{right}}+1}^{\mathsf{suf\text{-}ind}(i_{\text{right}})} v_j$$

where i_{right} is the index of the right-most leaf of the sub-tree rooted at the internal node.
- ℓ is the control bit of the left-most leaf in the sub-tree rooted at the internal node.
- p is the product (logical conjunction) of the control bits of all the leaves of the sub-tree rooted at the internal node.

Initialization. It is easy to see that by definition, the δ values at the leaves are the values in $\mathcal{V}$ if the control bit is 1, and $e_\star$ if the control bit it 0. Since leaves have empty right sub-trees, the γ value of a leaf is $e_\star$. By definition, the ℓ and p values at the leaves are the control bits $\mathcal{B}$. The suffix values are only calculated during the downstream phase and throughout the upstream phase, we simply set $\mathsf{suffix} = e_\star$. This shows that the 5-tuples at the leaves are consistent with the above semantics.

Upstream. We now inductively show that the 5-tuples are updated correctly during the upstream phase. The update to γ as $\gamma = \delta_1$ is consistent by definition. The suffix values are only calculated during the downstream phase and throughout the upstream phase, we simply set $\mathsf{suffix} = e_\star$. Since the left-most leaf of the left sub-tree rooted at a node is the same as the left-most leaf of the sub-tree rooted at the same node, the update $\ell = \ell_0$ is consistent. It is also easy to see that the product of the control bits of all the leaves of the sub-tree rooted at a node is the product of those in the left sub-tree rooted at the same node multiplied by the product of those in the right sub-tree rooted at the same node, and hence the update $p = p_0 p_1$ is consistent. Finally, we look at the update to δ. Recall that δ is the sum of the leaves of the sub-tree rooted at the internal node, starting from the left until and excluding the first value whose corresponding control bit is 0 (if it exists). It is helpful to think of δ as the "leading sum" (sum of values associated with the left-most sequence of control bits of the form $1111\ldots1$) of the sub-tree rooted at the internal node. It is easy to see that if the left sub-tree of an internal node contains a leaf with the control bit 0, then the leading sum of the sub-tree rooted at the internal node is the same as the leading sum of the left sub-tree of the node. Formally, this means that if $p_0 = 0$ (the product of the control bits in the left sub-tree is 0 if and only if one of the control bits is 0), then $\delta = \delta_0$. In the other case, where the left sub-tree has leaves all of whose control bits are 1, δ_0, the leading sum of the left sub-tree, would be the sum of the values of all the leaves in the left sub-tree, and the leading sum of the entire sub-tree rooted at the internal node would be the aggregate of δ_0 and the leading sum of the right sub-tree. Formally, this means that if $p_0 = 1$, then $\delta = \delta_0 \star \delta_1$. Thus, we have shown inductively that the 5-tuples are updated correctly during the upstream phase and are consistent with the semantics described above.

Downstream. What remains to be shown is that inductively the suffix values are updated correctly during the downstream phase. To begin, at the root, the suffix value is the suffix aggregation value of the right-most leaf of the entire tree, excluding its own value. Plugging in for the variables, at the root,

$$\mathsf{suffix} = \bigstar_{j=n+1}^{n} v_j = e_\star$$

which has been correctly set. We now proceed with the inductive claim. Suppose we have the 5-tuple at an internal node. We would like to update the suffix values of the left and right children of this node and proceed inductively. Since the right-most leaf of the right sub-tree of a node is the same as the right-most leaf of the sub-tree rooted at the same node, the update $\mathsf{suffix}_1 = \mathsf{suffix}$ is consistent. Consider the left sub-tree of the node. The suffix aggregation value of the that right-most leaf of the left sub-tree, excluding its own value, would include the leading sum of the right sub-tree of the node. If the right sub-tree had a leaf with the control bit 0, then the suffix aggregation value of the right-most leaf of the left sub-tree of the node would simply be the leading sum of the right sub-tree of the node. Formally, this means that if $p_1 = 0$ (the product of the control bits in the right sub-tree is 0 if and only if one of the control bits is 0), then $\mathsf{suffix}_0 = \gamma$. If the right sub-tree had no leaf with the control bit 0, then the suffix aggregation value of the right-most leaf of the left sub-tree of the node would be the aggregation of the leaves of the right sub-tree of the node as well as the suffix aggregation value of the right-most leaf of the right sub-tree of the node. Formally, this means that if $p_1 = 1$, then $\mathsf{suffix}_0 = \gamma \star \mathsf{suffix}$. Thus, we have shown inductively that the suffix values are updated correctly during the downstream phase and are consistent with the semantics described above.

Output. We now finally show that $\mathcal{V}_{\mathrm{suf}}$ is calculated correctly. Indeed, by our previous arguments,

$$\mathsf{suffix}_i = \bigstar_{j=i+1}^{\mathsf{suf\text{-}ind}(i)} v_j$$

Hence

$$v_i' = v_i \star \mathsf{suffix}_i = \bigstar_{j=i}^{\mathsf{suf\text{-}ind}(i)} v_j$$

as required. □

B.2 Full Aggregation

Initialization. We initialize the 8-tuples for the leaves as $(e_\star, v_i, e_\star,\ \delta_i, e_\star, e_\star, b_i, b_i)$, where

$$\delta_i = \begin{cases} e_\star & b_i = 0 \\ v_i & b_i = 1 \end{cases}$$

Upstream. Consider an internal node with left and right children having the associated 8-tuples $(\alpha_0, \beta_0, \gamma_0, \delta_0, , \mathsf{suffix}_0, \ell_0, p_0)$ and $(\alpha_1, \beta_1, \gamma_1, \delta_1, \mathsf{prefix}_1, \mathsf{suffix}_1, \ell_1, p_1)$ respectively. Then, we calculate the 8-tuple of the internal node as

$$\alpha = \beta_0$$

$$\beta = \begin{cases} \beta_1 & p_1 = 0 \\ \beta_0 \star \beta_1 & p_1 = 1 \end{cases}$$

$$\gamma = \delta_1$$

$$\delta = \begin{cases} \delta_0 & p_0 = 0 \\ \delta_0 \star \delta_1 & p_0 = 1 \end{cases}$$

$$\mathsf{prefix} = e_\star, \mathsf{suffix} = e_\star, \ell = \ell_0, p = p_0 p_1$$

Downstream. Consider an internal node with an associated 8-tuple $(\alpha, \beta, \gamma, \delta, \mathsf{prefix}, \mathsf{suffix}, \ell, p)$. Then, we calculate the prefix and suffix values for its left and right children as

$$\mathsf{prefix}_0 = \mathsf{prefix}$$

$$\mathsf{prefix}_1 = \begin{cases} e_\star & \ell_1 = 0 \\ \alpha & \ell_1 = 1, p_0 = 0 \\ \mathsf{prefix} \star \alpha & \ell_1 = 1, p_0 = 1 \end{cases}$$

$$\mathsf{suffix}_0 = \begin{cases} \gamma & p_1 = 0 \\ \gamma \star \mathsf{suffix} & p_1 = 1 \end{cases}$$

$$\mathsf{suffix}_1 = \mathsf{suffix}$$

Output. We calculate the aggregation of $\mathcal{V}$ as $\mathcal{V}_{\text{agg}} = \left[v'_1, v'_2, \ldots, v'_n\right]$, where

$$v'_i = \mathsf{prefix}_i \star v_i \star \mathsf{suffix}_i$$

and prefix_i and suffix_i are the prefix and suffix values in the 8-tuple associated with the leaf with value v_i at the end of the downstream phase.