ORQ: Complex Analytics on Private Data with Strong Security Guarantees

Eli Baum[1], Sam Buxbaum[1], Nitin Mathai[a,2], Muhammad Faisal[1], Vasiliki Kalavri[1], Mayank Varia[1] and John Liagouris[1]

[1] Boston University, Boston, MA, USA
[2] University of Texas at Austin, Austin, TX, USA

Abstract. We present ORQ, a system that enables collaborative analysis of large private datasets using cryptographically secure multiparty computation (MPC). ORQ protects data against semi-honest or malicious parties and can efficiently evaluate relational queries with multi-way joins and aggregations that have been considered notoriously expensive under MPC. To do so, ORQ eliminates the quadratic cost of secure joins by leveraging the fact that, in practice, the structure of many real queries allows us to join records and apply the aggregations "on the fly" while keeping the result size bounded. On the system side, ORQ contributes generic oblivious operators, a data-parallel vectorized query engine, a communication layer that amortizes MPC network costs, and a dataflow API for expressing relational analytics — all built from the ground up.

We evaluate ORQ in LAN and WAN deployments on a diverse set of workloads, including complex queries with multiple joins and custom aggregations. When compared to state-of-the-art solutions, ORQ significantly reduces MPC execution times and can process one order of magnitude larger datasets. For our most challenging workload, the full TPC-H benchmark, we report results entirely under MPC with Scale Factor 10 — a scale that had previously been achieved only with information leakage or the use of trusted compute.

1 Introduction

Secure multiparty computation (MPC) [Lin20] is a cryptographic technique that distributes trust across non-colluding parties, which perform a computation collectively. Thanks to its decentralized nature, it offers a robust solution that protects data both from individual parties and from powerful adversaries, who may have compromised a subset of the parties.

To avoid leaking information about the data, MPC programs are *oblivious*, i.e., they perform the same operations and memory accesses for all inputs of the same size. In practice, oblivious execution hides access patterns and the data distribution by generating worst-case outputs. For example, a secure MPC filter on a table with n records returns a table of the same size. This guarantees that the filter selectivity remains hidden from computing parties and external adversaries. However, hiding the data distribution for binary operators incurs a quadratic blowup in the worst case. An oblivious relational

Extended version of [BBM+25] (SOSP 2025).

E-mail: elibaum@bu.edu (Eli Baum), sambux@bu.edu (Sam Buxbaum), nitinm@cs.utexas.edu (Nitin Mathai), mfaisal@bu.edu (Muhammad Faisal), vkalavri@bu.edu (Vasiliki Kalavri), varia@bu.edu (Mayank Varia), liagos@bu.edu (John Liagouris)

[a]Work done while at Boston University

join on two input tables with n records will return the Cartesian product of size n^2. The problem gets worse for large n and multi-way join queries, due to a cascading effect: in general, a naive oblivious evaluation of a query with k joins has $O(n^{k+1})$ time and space complexity.

Secure join queries have been extensively studied in the context of peer-to-peer MPC, where data owners also act as computing parties [VSG+19, PKY+21, FCH+24, BHE+18, BPH+20, BEE+17, PHP+24, LWDY24, WY21]. In this setting, performance can be improved by introducing controlled leakage [NH12, BHE+18, BPH+20], offloading computation to trusted parties [VSG+19], or by applying optimizations tailored to specific data ownership schemes [FCH+24, WY21, LWDY24, PKY+21, BEE+17]. For example, SecretFlow [FCH+24] assumes the participation of exactly two data owners and Senate [PKY+21] expects each party to be a data owner who contributes an entire table to the analysis. Despite their particular differences, all of these works are restricted to a single (often custom) MPC protocol and threat model, offload expensive operations to data owners for plaintext execution in their trusted compute, and do not scale beyond a small number of participants.

To lift the restriction on the number of data owners and the need for trusted resources, MPC has also been deployed in the outsourced setting [Moz19, App21, CGB17, Bos24, HZF+22, KVH+20, AHK+23, MZF+23, LKFV23, FZL+23, RZCGP24], where computation is performed by a small set of untrusted servers. Although promising, this approach makes oblivious join queries even more challenging, as servers never get access to plaintext data and cannot perform operations in the clear. Prior works on outsourced systems for relational analytics suggest that avoiding the cascading effect is inherently incompatible with strong security guarantees: Secrecy [LKFV23] ensures no information leakage but its join operator has quadratic cost, while Scape [HZF+22] reduces the complexity to $O(n \log^2 n)$ but leaks the join result size to the servers.

Core problem. Making secure analytics practical in the outsourced setting requires tackling a hard open problem: *how to efficiently evaluate queries that involve joins where both input tables may contain duplicates.* This is the crux of the cascading effect and becomes the norm when multiple data owners contribute data to a joint analysis. To evaluate such queries, all prior MPC works that support fully oblivious execution either compute the Cartesian product of the input tables (which is expensive), enforce a subquadratic upper bound on the join output (which silently drops joined records), or leak the join result size (which is insecure). In fact, the problem is not specific to MPC but manifests itself in the broader area of oblivious analytics. When it comes to joins on duplicate keys, even state-of-the-art approaches for trusted hardware either require an upper bound on the join result [ZDB+17, DLP+20] or introduce volume leakage [MWD+25].

This paper describes the design and implementation of Orq, an *Oblivious Relational Query engine* that enables efficient analytics in the outsourced MPC setting with strong security guarantees. Orq takes a holistic approach that considers the entire workload and supports a broad class of analytics without suffering from the limitations of prior approaches. We make the observation that all relational queries used in prominent MPC works (even those that include multi-way joins on potentially duplicate keys) produce *results whose size is worst-case bounded by the input size*; that is, there exists an upper bound $O(n)$ on the query output that is independent of the data distribution. By analyzing 31 real and synthetic workloads we collected from the standard TPC-H benchmark [Tra24] and prior MPC works, we find that the cascading effect is avoidable in all of them, even in the queries that have been used to pinpoint the problem and motivate the need for leakage [BHE+18]. Orq leverages the data-independent upper bound on the query result size to achieve fully oblivious tractable execution — entirely under MPC — with no leakage and without relying on trusted compute.

A new scalable approach to oblivious analytics. In a typical MPC scenario, data owners would only agree to participate in analyses that return aggregated results (to preserve their privacy), and, oftentimes, the aggregation function is amenable to partial evaluation. Driven by this insight, Orq's core novelty is a technique that avoids materializing the Cartesian product of joins by decomposing and applying the aggregation eagerly (whenever possible) to bound the size of intermediate results. In practice, Orq performs a series of composite join-aggregation operations resembling a MapReduce-style evaluation [ORSS08]. To achieve competitive performance, Orq generalizes recent oblivious shuffling and sorting techniques [AHI+22, PRRS24] to different MPC protocols, optimizes them for tabular data (to avoid redundant column permutations), and integrates them with a butterfly-style control flow [Goo11, BA12] (to further reduce communication rounds between parties). It evaluates all queries we have collected from prior MPC works with *asymptotic cost equal to that of sorting the input tables*: $O(n \log n)$ operations, $O(\log n)$ rounds, and $O(n)$ memory, where n is the total number of input records.

Contributions. We make the following contributions:

- We develop Orq, a system that enables complex analytics on large private datasets using cryptographically secure MPC. Orq provides a novel system runtime that employs data-parallel vectorized execution, a high-level dataflow API for composing secure analytics on tabular data, and an efficient communication layer that amortizes MPC costs in LAN and WAN settings.

- We design an oblivious join-aggregation operator that supports all join types (inner, outer, semi- and anti-join), a broad class of aggregation functions, and composition with itself or other operators (e.g., user-defined filters) to form *arbitrary* data pipelines. Using this operator and known transformations from relational algebra, Orq can evaluate all queries we have collected from prior MPC works, without suffering from the quadratic blowup of intermediate results.

- We design all Orq operators to make black-box use of MPC primitives (e.g., $+, \times, \texttt{div}, \oplus, \wedge$). As such, they can be easily instantiated with *any* MPC protocol to support use cases with different threat models and security requirements. To showcase generality, we instantiate *Orq* with three protocols: two semi-honest (for honest and dishonest majorities) and one maliciously secure protocol in the honest-majority setting.

- We present a comprehensive experimental evaluation on diverse workloads, including the *complete* TPC-H benchmark (for the first time under MPC)[1] and nine queries from prior works inspired by real use cases. We show that Orq significantly outperforms state-of-the-art relational MPC systems while also scaling complex analytics to one order of magnitude larger inputs.

Additionally, we contribute data-parallel vectorized implementations of oblivious shuffling, quicksort, and radixsort that achieve better concrete performance than existing approaches. For oblivious sorting, we report results with up to half a billion rows — 10× larger than the best published results [AHI+22, AFO+21] (whose implementations are not publicly available). We have released Orq as open-source at:

https://github.com/CASP-Systems-BU/orq.

[1] We do not yet support substring operations.

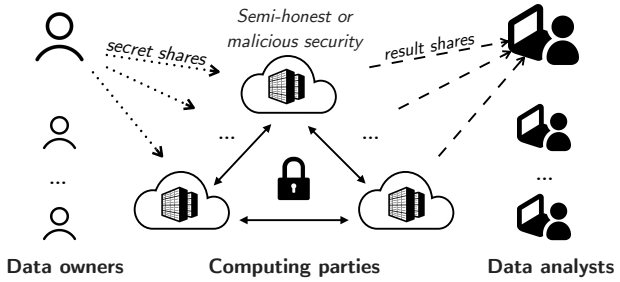

Figure 1: ORQ targets the typical outsourced setting that uses a small set of computing parties (whose number depends on the MPC protocol) to support any number of data owners and analysts.

1.1 Overview

This extended version of ORQ provides further background material and additional benchmarks. In Section 2, we introduce the ORQ system, its supported workloads, and its security guarantees. Section 3 provides readers with a gentle introduction to secure computation, with a focus on multiparty protocols. Next, Section 4 introduces ORQ's major operators, while Section 5 describes our oblivious sorting framework. Section 6 illustrates our new composite join operator. Finally, Section 7 describes our implementation, and Section 8 evaluates ORQ on a series of microbenchmarks and real-world queries. We discuss related work in Section 9. We also include appendices with more technical details on oblivious shuffle (§A), sorting (§B and §C), and joins (§D). Bandwidth measurements for the experiments in §8 are included in Appendix E.

2 The ORQ System

ORQ targets the outsourced setting in Figure 1 that has recently gained popularity in industry [Moz19, App21, MZF+23, HZF+22, KVH+20], academia [CGB17, DRPS22, RZCGP24, LKFV23, FZL+23], and non-profit organizations [Uni23]. In our setting, *data owners* distribute *secret shares* of their data to a set of untrusted non-colluding *computing parties*, in practice, servers managed by different infrastructure providers. Computing parties execute a query on the input shares, following a specified MPC protocol, and send shares of the result only to the designated *data analysts*. ORQ currently supports three MPC protocols and can be instantiated with semi-honest or malicious security. We describe ORQ's secret-sharing techniques and protocols in §3.1–2.3.

ORQ is designed for the typical outsourced setting with no access to a Trusted Compute Base (TCB) [LKFV23, HZF+22], but can also be deployed on premises in scenarios where data owners want to participate in the computation [VSG+19, BEE+17, PKY+21] (and in that case it could use optimizations from prior work that leverage trusted resources). Naturally, ORQ also supports the scenario in Flock [KTKP24], where a single data owner wants to securely outsource some computation to the cloud.

2.1 Supported workloads

ORQ supports a rich class of relational workloads, allowing an arbitrary composition of the following operators to form query plans: `SELECT`, `PROJECT`, `INNER_JOIN`, `(RIGHT / LEFT / FULL)-OUTER_JOIN`, `SEMI-JOIN`, `ANTI-JOIN`, `GROUP BY`, `DISTINCT`, `ORDER BY` and `LIMIT`. It also supports all common aggregations (`COUNT`, `SUM`, `MIN`, `MAX`, `AVG`) and user-defined aggregations, which can be constructed with its secure primitives: addition, multiplication,

bitwise operations, comparisons, and division with private or public divisor. Orq does not impose any restrictions on the database schema, such as the existence of integrity constraints (e.g., primary or foreign keys), but analysts can leverage these constraints, if they exist, to improve execution performance.

What can Orq compute efficiently? Orq's focus is on complex, multi-way join-aggregation queries that have so far been considered impractical in the outsourced MPC setting [HZF+22, AHK+23]. Specifically, Orq can efficiently compute acyclic conjunctive queries [Yan81] that include: (i) only one-to-many joins, i.e., joins where at least one input has unique keys, or (ii) many-to-many joins (with duplicate keys in both inputs) as long as there exists a decomposable (or algebraic) aggregation function [YGI09, GBLP96], and any group-by keys appear in a single input table. Interestingly, all queries we have collected from prior relational MPC works [VSG+19, LKFV23, PKY+21, BHE+18, FCH+24, WY21, BEE+17, LWDY24, BPH+20] fall in one of these two categories.

We note that many-to-many joins (also known as cross joins) are particularly common in collaborative data analysis because integrity constraints (e.g., the existence of unique join keys) may not be public. Moreover, even if integrity constraints are known, it is hard to enforce them outside MPC, unless data owners rely on a trusted third party or are willing to reveal some information about their data.

What can Orq not compute efficiently? There are queries outside this class that are inherently difficult to perform under MPC, and for these queries Orq falls back to an oblivious $O(n^2)$ join algorithm, like prior work. Examples of such queries (in SQL syntax) are: `SELECT COUNT(*) FROM A,B,C WHERE A.X=B.Y AND B.Y=C.W AND C.W=A.Z` (cyclic), or `SELECT COUNT(*) FROM A,B WHERE A.X=B.Y AND A.W>B.Z`, when `A.X` contains duplicates (aggregation across tables).

2.2 Programming model

Orq users write queries in a dataflow-style API, similar to the ones provided by Apache Spark [ZXW+16] and Conclave [VSG+19]. Relational operators are modeled as transformations on input tables and can be chained to construct complex queries. Users submit their queries to the Orq engine, which compiles them into secure MPC programs for the computing parties.

Listing 1 shows an implementation of TPC-H Q3 in the Orq API (see the benchmark specification [Tra24] for the SQL definition). The query combines filters, joins, and aggregations to compute the shipping priority and total revenue of customer orders that had not been shipped as of a given date.

Filters (lines 5–7) expect logical predicates that are constructed with Orq secure primitives (e.g., $\wedge$, $=$, $\neq$, $\geq$, $\leq$). Joins expect a key and an optional list of aggregation functions (each with an input and output column) that will be applied on the same key as the join. Line 13 joins tables `C` and `O` on column `CustKey`. The result is a new (anonymous) table that is then joined with table `LI` on column `OrdKey`, in line 14. The join propagates `OrdDate` and `Priority` values from the left input to the output by applying the function `copy`.

The `aggregate` operator (line 17) expects a list of columns as grouping keys (`{"OrdKey", "OrdDate", "Priority"}`) and a list of functions along with their input and output columns (`{"Revenue", "TotalRevenue", sum}`). Orq can apply multiple aggregation functions in a single control flow (see §6.2) and can also update columns in place (e.g., `{"C", "C", sum}`) to reuse memory. Orq provides a set of built-in aggregation functions (e.g., `count`, `sum`, `min`, `max`, and `avg`) but also allows users to define custom ones.

```
// Initialize C (Customers), O (Orders), LI (Lineitem)
...

// Apply filters
C.filter("MktSegment" == SEGMENT);
O.filter("OrdDate" < DATE);
LI.filter("ShipDate" > DATE);

// Calculate the revenue
LI["Revenue"] = LI["Price"] * (100 - LI["Discount"]) / 100;

// Joins and group-by with aggregation
auto RES = C.inner_join(O, {"CustKey"})
            .inner_join(LI, {"OrdKey"}, {
                {"OrdDate",  "OrdDate",  copy},
                {"Priority", "Priority", copy}})
            .aggregate({"OrdKey", "OrdDate", "Priority"}, {
                {"Revenue", "TotalRevenue", sum}})
            .project({"OrdKey", "TotalRevenue",
                      "OrdDate", "Priority"})
            .sort({{"TotalRevenue", DESC}, {"OrdDate", ASC}})
            .limit(10);
```

Listing 1: TPC-H Q3 with the ORQ API

Our rationale for exposing a dataflow-style API in ORQ instead of a SQL interface was motivated by a recent critique of SQL [NL24]. We also found that multi-way join queries with nested sub-queries written with the ORQ API were often more concise and easier to understand than their SQL counterparts. For this paper, we implemented 31 queries of varying complexity in a few hours.

2.3 Threat models and security guarantees

Like in prior works [VSG+19, PKY+21, LKFV23, FZL+23], data owners and analysts who want to use ORQ must agree beforehand on the computation to execute. The query and data schema are public and also known to the computing parties. To ensure data privacy, ORQ uses end-to-end oblivious computation and protects all input, intermediate, and output data — including the sizes of intermediate and final results — throughout the analysis.

Threat models. MPC protocols protect data against computing parties and external adversaries who may control T out of N computing parties, where $T < N$. ORQ can withstand adversaries who can be *semi-honest* ("honest-but-curious") or *malicious*. A semi-honest adversary passively attempts to break privacy without deviating from the protocol (by monitoring the state of the parties it controls, e.g., via inspection of messages and access patterns). A malicious adversary actively attempts to break privacy or correctness and can deviate arbitrarily from the protocol for the parties it controls.

Supported protocols. ORQ operators are protocol-agnostic, that is, they use the underlying MPC functionalities $(+, \times, \oplus, \wedge)$ as black boxes [DN03]. This way, they can be instantiated for any MPC protocol by simply replacing the corresponding functionalities. In its current version, ORQ supports three state-of-the-art protocols with different threat models and guarantees: (i) the semi-honest ABY protocol [DSZ15] for dishonest majorities

($T = 1$, $N = 2$), (ii) the semi-honest protocol by Araki et al. [AFL+16] for honest majorities ($T = 1$, $N = 3$), and (iii) the malicious-secure Fantastic Four protocol [DEK21] for honest majorities ($T = 1$, $N = 4$).

Security guarantees. Orq inherits the security guarantees of its underlying MPC protocols, ensures a fully oblivious control flow, and always operates over secret-shared data. It provides (i) *privacy*, which means that computing parties and adversaries do not learn anything about the data, and (ii) *correctness*, which means that all participants are convinced that the computation result is accurate. Our malicious-secure protocol provides security *with abort*, meaning that honest parties halt the computation if they detect malicious behavior (to protect data privacy). That said, Orq can easily be amended to support robust execution if desired.

3 Background on secure computation

The field of secure computation investigates efficient ways to execute *public programs* on *private data*. We wish to run computations in such a way that nothing more than the output is revealed: inputs and intermediate values remain hidden. There are three general approaches to solving this problem:

1. **Trusted Execution Environments** (TEEs) provide security via hardware protections. Information is encrypted at rest, and only decrypted within a secure enclave which performs the actual computations.

2. **Multiparty Computation** (MPC) distributes trust among multiple distinct parties and assumes that an adversary can only compromise a strict subset of the parties. Programs are decomposed into distributed protocols, where parties only exchange masked values which reveal nothing about the data.

3. **Fully Homomorphic Encryption** (FHE) bases secure computation on encryption schemes which are designed such that operations can be performed *directly* on ciphertexts and fully outsourced to an untrusted party, with clients performing only (relatively) cheap encryption and decryption operations.

The current state of affairs is such that TEEs provide the lowest overhead for secure computation — but carry with them the greatest risks. FHE represents the holy grail of secure computation, but current approaches are orders of magnitude slower. MPC represents a middle ground, where trust assumptions are minimal but protocols are efficient enough to evaluate real-world workloads. We do not address in this paper the related, but much more challenging, goal of hiding the program itself.

3.1 Approaches to multiparty computation

There are two main kinds of multiparty computation: *garbled circuits* and *secret sharing*. We focus exclusively on the latter in Orq, but for completeness we provide a brief overview of garbled circuit protocols. The *garbler* party "encrypts" the truth table for each gate in the computation circuit (represented as a directed acyclic graph of logic gates) in such a way that the *evaluator* party can obliviously execute the circuit. Garbled circuit protocols execute in a constant number of rounds but typically have higher bandwidths: for every bit of data in a circuit, they must transmit multiple ciphertexts. While recent theoretical works bring this communication overhead down [LWYY25] at the expense of increased computation, garbling is usually not the practical approach for real-world workloads,

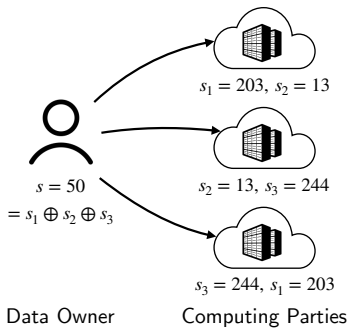

Figure 2: Example of replicated boolean secret sharing

where we deal with very wide circuits (i.e., vectorized inputs) and relatively low-latency connections between all parties.

By contrast, ORQ protects data using *secret sharing* [Sha79]. Given N computing parties, a secret ℓ-bit string s is encoded using randomized ℓ-bit shares $s_1, s_2, \cdots, s_N$ that individually are uniformly distributed over all possible ℓ-bit strings and collectively are sufficient to reconstruct the secret s. ORQ supports two types of secret sharing: *arithmetic* (for numbers) and *boolean* (for strings or numbers). In arithmetic sharing, the secret value s is encoded with N random shares such that $s = s_1 + s_2 + \cdots + s_N \bmod 2^\ell$, where mod handles overflow. Similarly, in boolean sharing, s is encoded such that $s = s_1 \oplus s_2 \oplus \cdots \oplus s_N$, where $\oplus$ is bitwise XOR. ORQ also provides efficient primitives to convert between the two representations without relying on data owners. In this paper, when not clear from context, we will use $[\![x]\!]$ to denote a secret sharing of x. Figure 2 shows an example: the data owner, with secret $s = 50$, randomly generates three secret shares $(203, 13, 244)$ and shares two with each of the three parties.

Data owners distribute secret shares to non-colluding computing parties, which in practice are ORQ servers running in different trust domains. Each computing party receives a strict subset of the shares (per secret) so that no single party can reconstruct the original data. Given random shares, parties can execute a secure MPC protocol to jointly perform arbitrary computations on the shares — as if they had access to the secrets — and end up with shares of the result. Operations on shares break down into a set of low-level secure primitives (e.g., $+, \times, \oplus, \wedge$), which are more expensive than the respective plaintext primitives because they require communication: for two ℓ-bit shares, secure bitwise AND requires exchanging $O(\ell)$ bits between computing parties. XOR is a local operation which parties can apply directly to their secret shares:

$$x = x_1 \oplus x_2 \oplus x_3 \qquad y = y_1 \oplus y_2 \oplus y_3$$
$$\mathcal{P}_1 : z_1 := x_1 \oplus y_1$$
$$\mathcal{P}_2 : z_2 := x_2 \oplus y_2$$
$$\mathcal{P}_3 : z_3 := x_3 \oplus y_3$$
$$(x_1 \oplus y_1) \oplus (x_2 \oplus y_2) \oplus (x_3 \oplus y_3) = x \oplus y$$

AND gates require communication, and thus a more complex protocol. ORQ supports multiple underlying MPC protocols (each of which targets a different threat model); Figure 3 and the example below discuss the AND protocol for our replicated three-party (semi-honest, honest majority) protocol [AFL+16].

So-called *replicated* secret-sharing protocols distribute multiple shares to each party. Observe that since all shares are random, each party still has no information about the secret: the third missing share always masks the true value. In the following discussion,

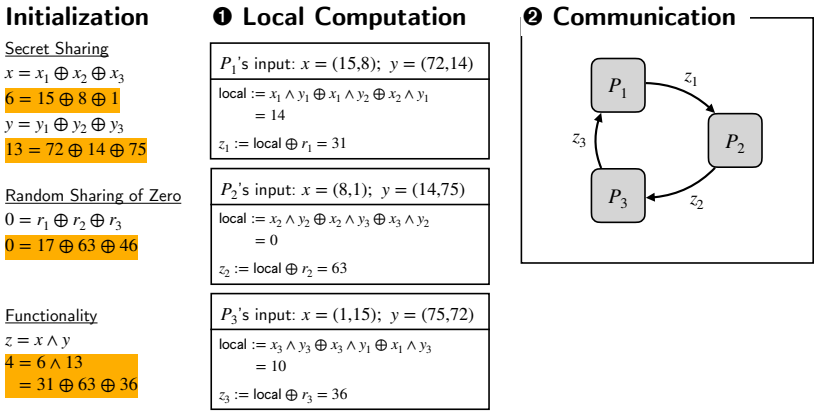

Figure 3: Overview of secure AND protocol by Araki et al. [AFL+16]. Example values are highlighted in orange. (1) Parties perform local computation on their shares. (2) They create a replicated sharing by sending one share to their neighboring party. The protocol requires a random sharing of zero, which can be generated non-interactively using pairwise pseudorandom generators with pre-shared keys.

we write ab to represent the componentwise (bitwise) AND $a \wedge b$.

$$\mathcal{P}_1 : (x_1, x_2) \qquad \mathcal{P}_2 : (x_2, x_3) \qquad \mathcal{P}_3 : (x_3, x_1)$$

To compute a sharing of $[\![x \wedge y]\!]$, we consider the expansion of the product:

$$\begin{aligned} x \wedge y = \quad & x_1y_1 \oplus x_1y_2 \oplus x_1y_3 \oplus \\ & x_2y_1 \oplus x_2y_2 \oplus x_2y_3 \oplus \\ & x_3y_1 \oplus x_1y_2 \oplus x_3y_3 \end{aligned}$$

Each party has sufficient information to compute a single share of the product.

$$\begin{aligned} \mathcal{P}_1 : z_1 &:= x_1y_1 \oplus x_1y_2 \oplus x_2y_1 \\ \mathcal{P}_2 : z_2 &:= x_2y_2 \oplus x_2y_3 \oplus x_3y_2 \\ \mathcal{P}_3 : z_3 &:= x_3y_3 \oplus x_3y_1 \oplus x_1y_3 \end{aligned}$$

However, while $\bigoplus_i z_i = x \cdot y$, the parties no longer possess a replicated sharing. Therefore, the final step is a communication phase, where parties mask their calculated shares with a random sharing of zero (i.e., $\bigoplus r_i = 0$) and send one to their neighbor. This communication is inherent to all MPC protocols and becomes the major bottleneck for large computations.

We did not yet discuss how to implement OR gates. Taking advantage of De Morgan's Law and the fact that bitwise negation is a local operation under boolean secret sharing (one party takes the negation of their shares), an OR gate has exactly the same cost as an AND gate: $x \vee y = \neg[(\neg x) \wedge (\neg y)]$. Together with the AND and XOR gates described above, this gives a complete boolean basis. When working over the ring of integers mod 2^k, the same protocols apply: XOR becomes the addition protocol; the AND gate is analogous to multiplication.

Historically, secret-sharing-based MPC was considered to be constrained by its higher number of communication rounds as compared to garbled circuits. However, ORQ demonstrates a viable approach to evaluate complex analytics in only $O(\log n)$ rounds; experimental evidence, even in the challenging WAN setting, confirms that we pay only a modest overhead compared to LAN.

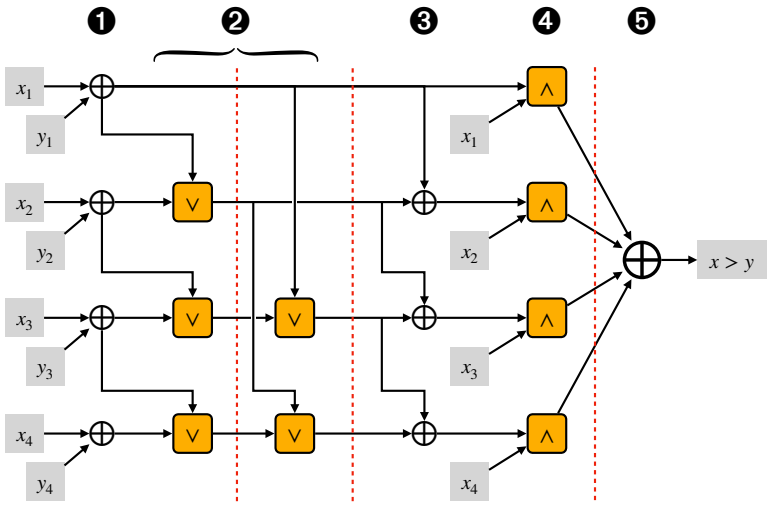

Figure 4: Example inequality circuit that computes $x > y$, where x, y are 4-bit unsigned integers. (1) Compute the bitwise XOR between x and y. (2) Use a prefix-OR circuit to identify the location of the first difference between x and y. (3) XOR this value with itself, offset by one, to generate a one-hot difference vector marking the location of the difference. (4) AND this value with the input x, to find the value of x at the bit-position where x and y first differ. If x is a one, then y must be a zero, so $x > y$. Otherwise, $x \leq y$. (5) Compute the parity of the outputs. **Orange gates** (AND, OR) require communication. XOR gates are local. Dashed red barriers denote dependent computational stages, and thus sequential rounds of communication. Gates in the same stage are independent and can be computed in parallel; the entire computation takes three rounds. OR gates have the same cost as AND gates.

3.2 From secure primitives to more complex operations

In this section, we discuss an example of a multiparty computation problem. We show how to compute the secure comparison of two secrets, by decomposing the problem into XOR and AND gates, as introduced above. Figure 4 shows a circuit representation of the procedure described in the text.

We assume x and y are 4 bits wide and secret shared in the boolean domain (although we show plaintext values below for simplicity). For illustrative purposes, we will set $x = 7$ and $y = 5$. In binary, these are:

$$\begin{array}{rcccc} x = & 0 & 1 & 1 & 1 \\ y = & 0 & 1 & 0 & 1 \end{array}$$

First, we compute a bitstring, $\mathsf{sameBits}$, which consists of all zeros up until the first bit that differs between x and y. This can be seen as $\mathsf{prefixOR}(\mathsf{x} \oplus \mathsf{y})$ (and therefore computed by any efficient parallel-prefix circuit, which decomposes into a series of XORs and ANDs).

$$\begin{array}{rcccc} x \oplus y = & 0 & 0 & 1 & 0 \\ \mathsf{sameBits} = & 0 & 0 & 1 & 1 \end{array}$$

At this point, observe that the LSB of $\mathsf{sameBits}$ gives (a secret sharing of) the equality predicate. Next, we calculate the expression: $(\mathsf{sameBits} \oplus (\mathsf{sameBits} \gg 1)) \wedge x$, which extracts the bit of x coincident with the first difference between x and y. If 1, x must be greater than y (since this implies y is 0 at that location, and the same as x at all prior

locations). Otherwise, x must be less than or equal to y.

$$\begin{aligned} \mathsf{sameBits} &= \quad 0 \quad 0 \quad 1 \quad 1 \\ \mathsf{sameBits} \gg 1 &= \quad 0 \quad 0 \quad 0 \quad 1 \\ \cdots \oplus \mathsf{sameBits} &= \quad 0 \quad 0 \quad 1 \quad 0 \\ \cdots \wedge x &= \quad 0 \quad 0 \quad 1 \quad 0 \end{aligned}$$

To move this bit to the LSB, we compute the XOR of all bits (implemented with `popcount` or a parity-check instruction) and place the result in the LSB.

$$[\![x > y]\!] = \quad 0 \quad 0 \quad 0 \quad 1$$

An important optimization in the design of efficient systems for MPC is the observation that independent messages within a stage can be batched together and sent in a single communication round. In the above computation, we do not need to perform eight sequential calls to the AND functionality. Instead, all run in parallel, and we communicate in a single round trip. This general structure also applies to the *vectorized* operations common in collaborative analytics.

To compute other inequalities, we use the standard identities: $a > b = \neg(a \leq b) = \neg(b \geq a)$. Further handling is necessary for signed integers; the result must be inverted if $\mathsf{sign}(x) \neq \mathsf{sign}(y)$ and $y < 0$. We omit a full description of this protocol for brevity and refer readers to our implementation for more details.

3.3 Obliviousness

Orq programs must be *oblivious*. A computation is oblivious if its control flow and observable side effects (such as data access patterns and timing) are input-independent, so that a computing party learns nothing about input, output, or intermediate values. Obliviousness prevents data reconstruction attacks [GLMP18, NKW15, BKM20, KKM+22, KKM+24] but has an inherent overhead compared to plaintext computation, as it requires using worst-case sizes and transforming all data-dependent conditionals (if-then-else statements) into data-independent ones. For example, consider the expression

```
if a > b then c = a else d = b
```

As written, a computing party (or adversary) could infer the relationship `a > b` by observing whether memory location `c` or `d` was written to. Such expressions exist in relational analytics, e.g., in filters and `CASE` statements. To avoid leaking information about the data, Orq transforms and evaluates this expression as shown below:

```
condition = (a > b);
c = (1 - condition) * c + condition * a;
d = (1 - condition) * b + condition * d;
```

We now write to both memory locations, irrespective of the condition, and follow no data-dependent branches. In practice, `a`, `b`, `c`, and `d` are secret-shared values, operator $>$ triggers a secure inequality protocol like the one described in §3.2, and operators $*, +, -$ correspond to secure functionalities provided by the underlying MPC protocol.

3.4 Dishonest-majority MPC in the preprocessing model

The two-party dishonest majority protocol [DSZ15] requires the generation of input-independent preprocessing material. We commonly refer to this preprocessing step as the *offline phase* and measure it separately. The *online phase* refers to the actual execution of the computation, after the inputs to the program are available.

Protocol 1: Beaver multiplication protocol

Input : $[\![x]\!], [\![y]\!] \in \mathcal{R}$
Output : $[\![z := x \times y]\!]$

$[\![a]\!], [\![b]\!], [\![c]\!] \leftarrow \mathsf{getTriple}(\mathcal{R})$
$\alpha \leftarrow \mathsf{open}(x + a)$
$\beta \leftarrow \mathsf{open}(y + b)$
return $[\![z]\!] := (\alpha \times [\![y]\!]) - ([\![a]\!] \times \beta) + [\![c]\!]$

Three kinds of preprocessing material, or correlated randomness, are required for our protocols: (1) boolean Beaver triples, for evaluating AND gates; (2) arithmetic Beaver triples, for evaluating multiplication gates; and (3) permutation correlations, for performing oblivious shuffle.

3.4.1 Beaver triples

Beaver [Bea91] made the simple but groundbreaking observation that correlated randomness could allow for the efficient evaluation of circuits in the online phase. For an arbitrary finite ring $\mathcal{R}$, a Beaver triple is a secret sharing of three values $a, b, c \in \mathcal{R}$ such that $a \times_{\mathcal{R}} b = c$, where $\times_{\mathcal{R}}$ represents the multiplication operation in the ring, and a, b are sampled randomly. We will often write such a triple as $([\![a]\!], [\![b]\!], [\![c]\!])$. Protocol 1 shows how we can "derandomize" a single Beaver triple to perform an arbitrary secret-shared multiplication.

Correctness follows directly:

$$\begin{aligned}\alpha[\![y]\!] - [\![a]\!]\beta + [\![c]\!] &= (x+a)[\![y]\!] - [\![a]\!](y+b) + [\![ab]\!] \\ &= [\![xy]\!] + [\![ay]\!] - [\![ay]\!] - [\![ab]\!] + [\![ab]\!] \\ &= [\![xy]\!]\end{aligned}$$

Security comes from the fact that a and b are uniform and thus perfectly mask x and y in the opened values α, β. The triple acts like a one-time pad; accordingly, each triple can be used only once (and, in fact, using a triple twice leaks information about the secret).

Example of rings. This general template applies over any (finite) ring.

- When $\mathcal{R} = \mathbb{Z}_{2^k}$, the integers mod 2^k, we have standard integer arithmetic. Generating such triples is expensive, but this ring is often the most natural one to express real-world problems in collaborative analytics. We implement the approaches of Gilboa [Gil99] and Doerner et al. [DHIM25].

- When $\mathcal{R} = \mathbb{F}_2$, we have the boolean triples. Here, operations are performed bitwise; addition is XOR ($\oplus$) and multiplication is AND ($\wedge$). These triples reduce down to a cryptographic primitive known as *random oblivious transfer*; we use the primitives provided by `libOTe` [RR24].

In the semi-honest setting, Beaver triples are generally generated non-interactively given *oblivious linear evaluations* (OLE) — correlations of the form $w_0 = x_1 \cdot y_0 + z_1$, where one party has (w_0, y_0) and the other has (x_1, z_1).[2] Given two OLEs, we can construct a

[2]The term "oblivious linear evaluation" refers to the view of P_1's randomness as defining a random line, and P_0's as a random point on that line. In a finite ring, exponentially many points lie on each line, and exponentially many lines pass through each point, so neither party can learn anything about the other's input.

Beaver triple with no further communication. Subscripts denote holders of each secret share.

$$\begin{aligned} \text{First OLE} \quad & s_0 = t_1 u_0 + v_1 \implies s_0 - v_1 = t_1 u_0 \\ \text{Second OLE} \quad & w_0 = x_1 y_0 + z_1 \implies w_0 - z_1 = x_1 y_0 \end{aligned}$$

$$\underbrace{(u_0 - x_1)}_{[\![a]\!]} \underbrace{(y_0 - t_1)}_{[\![b]\!]} = \underbrace{u_0 y_0 - u_0 t_1 - x_1 y_0 + x_1 t_1}_{[\![c]\!]}$$

The first and last terms of $[\![c]\!]$ can be computed locally by P_0 and P_1, respectively, and each OLE gives a secret sharing of the middle terms. The final shares of the Beaver triple are:

$$\begin{array}{cccc} & [\![a]\!] & [\![b]\!] & [\![c]\!] \\ P_0 : (& u_0, & y_0, & u_0 y_0 - s_0 - w_0 \quad) \\ P_1 : (& -x_1, & -t_1, & x_1 t_1 + v_1 + z_1 \quad) \end{array}$$

In the evaluation section (§8.7), we consider two protocols for generating base OLEs: the original one, due to Gilboa [Gil99], which directly emulates the schoolbook multiplication circuit using oblivious transfer, and has communication complexity $O(\ell^2)$; and a more recent construction by Doerner et al. [DHIM25] which uses the Chinese Remainder Theorem and a prime-field decomposition to evaluate a multiplication gate with only $O(\ell \log \ell)$ communication.

3.4.2 Permutation correlations

A thematically similar construction allows us to perform oblivious shuffles of large vectors. Let $\vec{x} \in \mathcal{R}^n$ be a vector of length n over some finite ring. A length-n *permutation correlation* is a pair of tuples $(\pi, \vec{B}_0), (\vec{A}, \vec{B}_1)$, where $\pi \in S_n$ (the symmetric group of size n) and $\vec{A}, \vec{B}_0, \vec{B}_1 \in \mathcal{R}^n$. They are correlated such that $\pi(\vec{A}) = \vec{B}_0 + \vec{B}_1$ for a suitable definition of the addition operation.

Say we have two parties, P_0 and P_1, and P_1 has the secret vector $\vec{x}$. We will show an example of how to use the correlation $(\mathcal{P}_0 : (\pi, \vec{B}_0), \mathcal{P}_1 : (\vec{A}, \vec{B}_1))$ to permute $\vec{x}$ by P_0's secret permutation π. First, P_1 masks the secret vector $\vec{x}$ with the random vector $\vec{A}$, and sends $\vec{\Delta} = \vec{x} - \vec{A}$ to P_0. P_1's share of the shuffled vector is $\vec{B}_1$. P_0, upon receiving $\vec{\Delta} = \vec{x} - \vec{A}$ from P_1, permutes $\vec{\Delta}$ by π to obtain $\vec{C}' = \pi(\vec{\Delta}) + \vec{B}_0$, and P_1's share of the shuffled vector is $\vec{C}'$. We can observe that the output is a secret sharing of $\pi(\vec{x})$, where the final step follows from the fact that $\pi(\vec{A}) = \vec{B}_0 + \vec{B}_1$.

$$\vec{B}_1 + \vec{C}' = \vec{B}_1 + \pi(\vec{x} - \vec{A}) + \vec{B}_0 = (\vec{B}_0 + \vec{B}_1) - \pi(\vec{A}) + \pi(\vec{x}) = \pi(\vec{x})$$

A single permutation correlation suffices for one party (the party holding the permutation π) to permute a secret-shared vector according to π. To perform an oblivious shuffle, where *neither* party knows the permutation that has been applied, we use two permutation correlations, π_0 and π_1, where each party acts as the sender (the party holding the permutation) in one of the correlations. The resulting permutation $\pi' = \pi_0 \circ \pi_1$, where $\circ$ represents composition, is random and hidden from both parties.

4 Oblivious building blocks

In this section, we review the oblivious primitives used as building blocks in Orq.

Protocol primitives. Protocol developers are expected to implement the basic cryptographic operations of a multiparty computation protocol: $+, \times, \oplus, \wedge$, along with utility primitives `open` and `secret_share`. From there, Orq provides implementations of comparison protocols ($=, \neq, >$), boolean circuits for addition (both ripple-carry and parallel-prefix) and division (non-restoring), along with conversion operators `a2b` and `b2a` to change between secret-sharing domains.

Table operations. To hide true table sizes, Orq tables (input, intermediate, or output) have a special *validity column* of secret-shared bits, encoding which rows are "valid" (1) or "dummy" (0). Dummy rows are those invalidated by oblivious operators and can also exist in input tables as a result of padding by data owners. Since valid bits are secret-shared, Orq servers cannot distinguish real from invalid rows and operate on worst-case table sizes during the entire execution. Validity columns are never exposed to parties; they are managed by Orq operators transparently. Invalid rows are masked and shuffled before opening to prevent leakage of "deleted" data.

Orq provides efficient implementations of core oblivious operators for relational tables. An oblivious **filter** applies a predicate to each row in the input table and outputs a secret-shared bit that denotes whether the row passes the filter. Oblivious **deduplication** (the "distinct" operator) is applied to one or more columns treated as a composite *key*. It obliviously sorts the table on these columns and then compares adjacent rows to mark the first occurrence of each distinct key with a secret-shared bit. Oblivious **multiplexing** is used in sorting, join, and aggregation operators. An arithmetic multiplexer $\texttt{Mux}(b, x, y)$ implements `b ? y : x` by evaluating $(1 - b) \cdot x + b \cdot y$, where $b \in \{0, 1\}$ and x, y are the elements to multiplex (boolean multiplexers are defined similarly).

Oblivious aggregation. Aggregation identifies groups of records based on a (composite) key and applies one or more aggregation functions to each group, updating the input table in place. Orq implements oblivious aggregation using sort followed by a butterfly-style network [Goo11, BA12, JKU11] which requires $O(n \log n)$ operations and messages, $O(\log n)$ communication rounds, and $O(n)$ space per party, where n is the cardinality of the input table. Protocol 2 shows the control flow of the aggregation, which resembles the Hillis-Steele network [HSJ86]. We give a proof of correctness in Appendix D.2. Orq also provides an implementation of the asymptotically-optimal Brent-Kung network [BK82].

The algorithm expects a table T sorted on column K (the grouping key) and compares keys at distance d, while doubling the distance after every iteration. Each step applies an aggregation function f to pairs of input values and multiplexes the result (in-place) into column G, based on whether the respective records belong to the same group ($b = 1$). For simplicity, the pseudocode assumes a single key column K and a single aggregation function f. In practice, however, Orq's operator expects one or more key columns and can evaluate more than one aggregation function *on the fly* (on the same or different columns) by reusing b (line 4). In §6, we show how Orq leverages this functionality to reduce the cost of joins followed by aggregation.

5 Sorting protocols for tabular data

We implement two oblivious sorting protocols, quicksort and radixsort, each competitive with the state of the art in terms of performance. Both algorithms rely on state-of-the-art oblivious shuffling [AHI+22, PRRS24], which we generalize to work with all MPC protocols in Orq.

In §5.1, we give background on the oblivious shuffling primitives upon which our sorting protocols are based. In §5.2, we describe the "base" version of each protocol as an in-place, ascending-order sorting protocol that requires unique keys. In §5.3, we describe a wrapper

Protocol 2: Aggregation Network (AggNet)

Input : Table T with n rows, group key column K, function f, input column A, result column G

Result : Table T with one value in G per unique K

$r.G \leftarrow r.A$ `// Copy input into result`

for $d = 1;\ d < n - d;\ d \leftarrow 2 \cdot d$ **do**

  for *each pair of rows* $(r_i,\ r_{i+d})$, $0 \leq i < n - d$ **do**

    $b \leftarrow r_i.K == r_{i+d}.K$ `// Compare keys`

    $g \leftarrow f(r_i.G,\ r_{i+d}.G)$ `// Aggregate`

    $r_{i+d}.G \leftarrow \texttt{Mux}(b,\ r_{i+d}.G,\ g)$ `// Update`

protocol that removes these restrictions and allows us to extract the sorting permutation applied to the input as a secret-shared permutation, which we then use to sort multiple columns in a table through our TableSort protocol in §5.4. In Appendix B, we compare our radixsort protocol in detail with the prior state of the art, and Appendix C discusses a probabilistic bound on the amount of preprocessing required for quicksort when using the two-party ABY [DSZ15] protocol.

Note on sorting complexity in MPC The lower bound on the complexity of general-purpose comparison-based sorting algorithms for input size n is $\Omega(n \log n)$ comparisons. In MPC, comparisons are expensive operations, typically requiring $\Omega(\ell)$ communication, where ℓ is the bitwidth of the elements. Hence, quicksort in MPC requires $\Omega(n \log n)$ comparison operations over ℓ-bit strings, for a total of $\Omega(\ell n \log n)$ bits of communication. Alternatively, sorting networks require $O(n \log^2 n)$ comparisons and $\Omega(\ell n \log^2 n)$ communication [Bat68]. Finally, radixsort [AHI+22] has either $O(\ell^2 n)$ or $O(\ell n \log n)$ communication depending on subtleties in the protocol that we discuss in Appendix B.

In the general case, where the plaintext vector might contain unique elements, our input space must have a bitwidth of at least $\ell = \Omega(\log n)$. Both quicksort and radixsort thus require $\Omega(n \log^2 n)$ communication, while sorting networks require $\Omega(n \log^3 n)$ communication. Hence, $\Omega(n \log^2 n)$ communication is the gold standard of general MPC sorting protocols.

5.1 Oblivious shuffling primitives

While a detailed discussion of our oblivious shuffling functionality is presented in Appendix A, here we outline the basic primitives and notation that are necessary to understand our sorting protocols.

We support two forms of secret sharing for permutations: *sharded permutations*, where the permutation is split into individually random permutations that compose to form the underlying permutation ($\langle \pi \rangle = \pi_n \circ \cdots \circ \pi_1$), and *elementwise permutations*, written as $[\![\pi]\!]$, where the destination indices of the permutation are secret shared as any other vector (either arithmetic or boolean). Additionally, we support `applyElementwisePerm` to apply an elementwise permutation to a vector, `invertElementwisePerm` for in-place inversion of the underlying permutation, and `convertElementwisePerm` to convert between a boolean and arithmetic secret sharing. We will generally use π to refer to random or unstructured permutations, and we will use σ to refer to sorting permutations.

Example. We briefly walk through an example of oblivious shuffling using a sharded permutation. Consider the two-party setting with parties P_0 and P_1, and let the input be a secret sharing of the vector $(1, 2, 3, 4)$. Each party generates a local random permutation:

P_0 generates $\pi_0 = (3, 1, 2, 4)$, and P_1 generates $\pi_1 = (1, 4, 3, 2)$. The parties will take turns applying their permutation to the secret-shared input obliviously (that is, without revealing to the other party which permutation has been applied) using the protocol by Peceny et al. [PRRS24]. The outcome is that first we apply $\pi_0 = (3, 1, 2, 4)$ to the vector $(1, 2, 3, 4)$, yielding $(2, 3, 1, 4)$, and then we apply $\pi_1 = (1, 4, 3, 2)$ to the intermediate vector to obtain the final shuffled vector $(2, 4, 1, 3)$.

5.2 Base sorting protocols

ORQ provides two base sorting protocols: an iterative version of quicksort and a radixsort protocol that combines features of Asharov et al. [AHI+22] and Bogdanov et al. [BLT14]. The protocols only sort in ascending order, and quicksort requires unique keys in the sorting column; these restrictions simplify the protocols and allow for more effective optimization.

5.2.1 Quicksort

Quicksort is typically considered to be the most efficient general sorting algorithm in practice. However, it typically has a recursive and data-dependent control flow, both of which make a naïve application of quicksort to the MPC setting problematic in both efficiency and security. The typical instantiation of the algorithm would incur $O(n)$ rounds of comparisons, which is prohibitively slow. To address this challenge, we devise an iterative control flow that applies all independent comparisons at a given recursion depth simultaneously, facilitating vectorization and reducing the number of rounds to $O(\log n \log \ell)$.

To benefit from the efficiency of quicksort in the MPC setting, we use the *shuffle-then-sort* paradigm by Hamada et al. [HKI+12, AFO+21] in which we obliviously shuffle the input vector first. As a result, the subsequent data-dependent sorting protocol only reveals the (meaningless) order of the shuffled vector. Thus, these random bits provably leak nothing about the data, and the control flow of the comparisons in quicksort is entirely independent of the input vector (this should not be confused with one-bit leakage protocols, such as the protocol by Zhang et al. [ZGY+24]).

The resulting protocol is described in Protocol 3 below. All sorting happens in place, and the protocol alternates between local computation and rounds of full vector comparisons. The protocol initializes a plaintext set of pivots S with the first element of the vector and iterates until S contains all elements in the vector. In each iteration, we compare every element at index i with one of the pivots: namely, the highest pivot index j such that $j \leq i$ via a function $j \leftarrow \texttt{prevPivot}(i)$ that can be computed locally by each party. We create a secret-shared vector $[\![\vec{c}]\!]$ of all previous-pivot elements, perform all comparisons between $\vec{x}$ and $\vec{c}$ in parallel, and omit comparisons for pivot elements (which would be compared with themselves).

After each round of secure comparisons, we reveal the result vector $\vec{r}$ and then partition the vector based on the desired pivot locations. Concretely, for each pivot p, we define a partitioning subroutine $\texttt{partition}([\![\vec{x}]\!], p, i, \vec{r})$ that takes as input the vector $[\![\vec{x}]\!]$, the start index p where the pivot is located, an end index i, and $\vec{r}$. The subroutine moves items in the subvector according to standard quicksort: all items less than the pivot, then the pivot itself, and all items greater than the pivot. It returns a set S' of up to three locations for the pivot position and the next-round pivots for the left and right halves. This step is entirely local since S' and $\vec{r}$ are common knowledge.

To improve concrete efficiency, we can represent S as a scaled indicator vector, $S_i = i \cdot \mathbb{I}[i \text{ is a pivot}]$; i.e., if index i is a pivot, index i of S contains i; otherwise, 0. Then, `prevPivot` is implementable via a single call to a prefix-max function (which computes a running maximum of a list). Likewise, the result of `partition` is no longer used to compute $S \cup S'$; instead, we update the new indices to contain their value, $\forall j \in S' : S_j = j$.

Protocol 3: Quicksort

input : $[\![\vec{x}]\!]$ containing unique elements
output : $[\![\vec{y}]\!] = \sigma([\![\vec{x}]\!])$

$[\![\vec{x}]\!] \leftarrow \texttt{shuffle}([\![\vec{x}]\!])$ // Oblivious shuffle
$S \leftarrow \{1\}$ // Initial set of pivots
while $|S| \neq n$ **do**
  $[\![\vec{c}]\!] \leftarrow [\,]$ // List of prev pivots to compare
  for i *from* 1 *to* n **do**
    $j \leftarrow \texttt{prevPivot}(i)$ // Previous pivot
    $[\![\vec{c}]\!]_i \leftarrow [\![\vec{x}]\!]_j$ // Set value to compare with
  $[\![\vec{r}]\!] \leftarrow \texttt{compare}([\![\vec{x}]\!], [\![\vec{c}]\!])$ // Perform the oblivious comparisons
  $\vec{r} \leftarrow \texttt{open}([\![\vec{r}]\!])$ // Reveal the results
  $i \leftarrow n$ // End index of current partition
  for $p \in \texttt{reversed}(S)$ **do**
    $S' \leftarrow \texttt{partition}([\![\vec{x}]\!], p, i, \vec{r})$ // Partition subvector around the pivot
    $S \leftarrow S \cup S'$ // Update pivot set
    $i \leftarrow p - 1$ // Update index
return $[\![\vec{x}]\!]$

To see (informally) why this is secure, consider the information revealed to the adversary through the opening of comparison results. After the oblivious shuffle, each comparison reveals the relationship between two randomly chosen (and unknown) elements in the input list. Combining the revealed comparison results from each iteration together amounts to revealing the permutation applied to the shuffled list. However, the random and unknown shuffling permutation perfectly masks this information, so the adversary learns nothing about the ordering of elements in the original list.

Example. We will now look at a toy example of the first iteration. Consider the input list $\vec{x} = (40, 50, 20, 80, 70, 30, 10, 60)$. The first step of the algorithm is to obliviously shuffle the list. Say the permutation applied is $\pi = (6, 1, 2, 3, 4, 8, 5, 7)$, which gives the shuffled vector $\pi(\vec{x}) = (50, 20, 80, 70, 10, 40, 60, 30)$. In the first iteration, we always choose the first element, 50, as our initial pivot and compare it with all elements, giving us the vector $(\pi(\vec{x}) \geq 50) = (1, 0, 1, 1, 0, 0, 1, 0)$. We then (locally) rearrange the elements according to the indicator vector, so values greater than or equal to 50 are moved to the back, yielding $(20, 10, 40, 30, 50, 80, 70, 60)$. The list is now correctly partitioned, where all elements preceding 50 are less than 50, and all elements following 50 are greater than 50. In the next iteration, our set of pivot elements will additionally contain 20 and 80, as they are the first elements in the newly created partitions.

We remark that Protocol 3 is only secure if all input elements in $[\![\vec{x}]\!]$ are unique, so that each comparison check in $\vec{r}$ is equally likely to be true or false based only on the random shuffle and not the data $\vec{x}$. We remove this limitation in §5.3 by making each element unique.

5.2.2 Radixsort

Radixsort sorts a vector one bit at a time, moving from the least to most significant bit. Bogdanov et al. [BLT14] permute the entire vector after each round of bit comparisons, while the later approach by Asharov et al. [AHI+22] composes the permutations bit by bit and applies them to the vector at the end, which is similar to what we do in TableSort at the table level (§5.4). For radixsort, we have found that a hybrid approach which

combines the eager technique by Bogdanov et al. [BLT14] with the shuffling primitives by Asharov et al. [AHI+22] reduces the number of rounds for a small increase in bandwidth and outperforms both protocols by up to 44% in practice. ORQ implements a vectorized version of this hybrid approach. We provide a detailed analysis in Appendix B.

The radixsort algorithm is shown in Protocol 4. Our protocol takes two inputs beyond the data vector: the total number of bits to sort, ℓ, and the number of least significant bits to skip, ℓ_s. The basic idea is simple: for each bit of the sorting key, we invoke the `genBitPerm` protocol of Asharov et al., which returns an elementwise sharing of the sorting permutation for that bit, σ_i. Although `genBitPerm` was proposed in the honest-majority three-party setting, we observe that it is actually agnostic to the setting and the number of parties, and it only makes black-box use of basic MPC primitives; therefore, it applies just as well in our dishonest-majority and malicious settings. We then apply the permutation to the full vector by invoking the `applyElementwisePerm` protocol.

Protocol 4: Radixsort

input : $[\![\vec{x}]\!]$, ℓ, ℓ_s
output : $[\![\vec{y}]\!] = \sigma([\![\vec{x}]\!])$

1 $[\![\vec{y}]\!] \leftarrow [\![\vec{x}]\!]$
2 **for** i *from* 1 *to* ℓ **do**
3 $\quad [\![\sigma_i]\!] \leftarrow$ `genBitPerm`$([\![\vec{y}]\!] \gg (i + \ell_s))$ `// Single bit sort`
4 $\quad [\![\vec{y}]\!] \leftarrow$ `applyElementwisePerm`$([\![\vec{y}]\!], [\![\sigma_i]\!])$ `// Permute according to sort`
5 **return** $[\![\vec{y}]\!]$

Example. We demonstrate the radixsort protocol with a small example. Consider the following vector of two-bit elements, written as bitvectors: $(01, 10, 01, 11, 10, 00, 10, 00)$. First, we isolate the least significant bit to obtain 10110000. We sort the bitvector to obtain 00000111, which gives us the applied permutation $\sigma_0 = (6, 1, 7, 8, 2, 3, 4, 5)$. We then apply σ_0 to the original input to sort the input by the least significant bit: $(10, 10, 00, 10, 00, 01, 01, 11)$. Next, we do the same to the most significant bit. We isolate the most significant bit: 11010001. Then, we sort the bit to obtain 00001111, giving us the applied permutation $\sigma_1 = (5, 6, 1, 7, 2, 3, 4, 8)$. We apply σ_1 to the current full vector to obtain our output: $(00, 00, 01, 01, 10, 10, 10, 11)$. We can see that the total permutation that has been applied from input to output is $\sigma_1 \circ \sigma_0$.

In the above example, we assumed that we could efficiently sort a single-bit vector, so we now demonstrate how that can be achieved. The insight (following from [AHI+22]) is that a single-bit sort can be decomposed into two prefix sums and a multiplexing. First, we negate every bit in the vector, and a prefix sum of the result yields a vector of destination indices for each value that was original a zero. Consider the example in the prior paragraph of the least significant bit: 10110000. We invert each bit to obtain 01001111, and a prefix sum yields $(0, 1, 1, 1, 2, 3, 4, 5)$. This matches the permutation σ_0 for indices where the original bit was a zero. We apply another prefix sum to the original (non-negated) vector to obtain $(1, 1, 2, 3, 3, 3, 3, 3)$. Here, each element corresponding to a one in the input is the destination index of that element if we index starting from the first position that will contain a one. That is, we add 5 (the number of zeros, obtained from the last element of the first prefix sum) to all elements, and we obtain $(6, 6, 7, 8, 8, 8, 8, 8)$. Finally, we multiplex between the two vectors, so if bit i in the input is a zero, we choose the i^{th} element from the first list, and if bit i is a one, we choose the i^{th} element from the second list. The output is the permutation $\sigma_0 = (6, 1, 7, 8, 2, 3, 4, 5)$.

5.3 Wrapper sort protocol

We now describe a general sorting wrapper protocol, VectorSort, that takes as input a boolean secret-shared vector $[\![\vec{x}]\!]$ to sort, the order in which to sort (either ascending (`ASC`) or descending (`DESC`)), and a base sorting protocol (either quicksort or radixsort). As output, it returns the sorted vector $[\![\vec{y}]\!] = \sigma([\![\vec{x}]\!])$ and an elementwise sharing of the sorting permutation $[\![\sigma]\!]$. The wrapper protocol performs two important operations: it pads the input before sorting, and it extracts the sorting permutation after sorting

Input padding. Following Hamada et al. [HKI+12], for each element of the input vector $\vec{x}$, we append its index. This padding ensures that the elements are unique, that we can extract the sorting permutation, and that the sort is *stable* in the sense that if $x_i = x_j$ with $i < j$, then x_i will remain before x_j in the sorted vector. Concretely, to sort an input vector $[\![\vec{x}]\!]$ in ascending order, we form the concatenated string $[\![x'_i]\!] = [\![x_i]\!] \,||\, [\![i]\!]$, with the sharing of i being a local operation using a protocol `publicShare`, which secret-shares a publicly known vector without communication. To ensure that all n elements in $\vec{x'}$ are unique, we must add at least $\lceil \lg n \rceil$ bits of padding; our implementation always adds 32 bits of padding.

Permutation extraction. After the base sort of $\vec{x'}$ is completed, the i^{th} element in the input vector, with the padded value i, winds up at position σ_i. We then extract the sorting permutation from the padding bits so that we can apply it to other columns in a table; to the best of our knowledge, no prior works have detailed how to extract the sorting permutation from the padding. We separate the sorted list $\vec{y'}$ into the data $\vec{y} = \sigma(\vec{x})$ and a permutation π. The padding bits contain the result of applying the sorting permutation σ to the identity permutation, so $\pi = \sigma(\mathbb{I}_n)$, where $\mathbb{I}_n$ represents the identity permutation $(1, \ldots, n)$. By Fact 1 (see Appendix A), $\sigma(\mathbb{I}_n) = \mathbb{I}_n \circ \sigma^{-1} = \pi$, so $\sigma = \pi^{-1}$. Hence, we obtain the sorting permutation σ by inverting π using the `invertElementwisePerm` protocol.

Descending order. Sorting an input vector $\vec{x}$ in descending order can be done similarly. As a thought experiment: it almost works to use the ascending-sort protocol above and locally reverse the output vector, except it breaks stability by reversing the tiebreak condition. That is, if $x_i = x_j$ and $i < j$, then ascending-sort preserves stability, but the local reverse ensures that x_i always comes after x_j instead.

To solve this issue, we negate the padding for each element. That is, each secret-shared element $[\![x_i]\!]$ now gets padded to $[\![x'_i]\!] = [\![x_i]\!] \,||\, [\![-i]\!]$. Since we are attaching the padding bits as the least significant bits, we need any two values $-i$ and $-j$ where $i < j$ to satisfy $-i > -j$ even when interpreted as an unsigned integer. Our implementation uses $\pm\{1, \ldots, n\}$ for the padding.

Putting it all together. We write the general sorting wrapper in Protocol 5 and summarize it here. We start by padding the input $[\![\vec{x}]\!]$ with $\mathbb{I}_n$ in order to perform an ascending sort for input size n, or with $-1 \cdot \mathbb{I}_n$ to sort in descending order. The resulting padded input vector $[\![\vec{x'}]\!]$ is fed into the base quicksort or radixsort protocol from §5.2, which results in the padded sorted vector $[\![\vec{y'}]\!]$. Finally, we locally split $[\![\vec{y'}]\!]$ into the unpadded result $[\![\vec{y}]\!]$ and a permutation $[\![\pi]\!]$, and invert $[\![\pi]\!]$ to obtain the sorting permutation $[\![\sigma]\!]$. If sorting in descending order, before inverting $[\![\pi]\!]$, we convert it from boolean to arithmetic sharing to negate each element in the permutation. We return both $[\![\vec{y}]\!]$ and $[\![\sigma]\!]$.

Next, we describe how to use Protocol 5 to efficiently sort a table with multiple columns.

Protocol 5: VectorSort

input : $[\![\vec{x}]\!]$, order $\in \{\texttt{ASC}, \texttt{DESC}\}$, sort $\in \{\texttt{quicksort}, \texttt{radixsort}\}$
output : $[\![\vec{y}]\!] = \sigma([\![\vec{x}]\!])$, $[\![\sigma]\!]$

1 $\vec{p} \leftarrow \mathbb{I}_n$ // Padding is the identity permutation
2 **if** *order* == *DESC* **then**
3 $\quad \forall i \in [n], p_i \leftarrow -p_i$ // Negate padding if sorting in reverse
4 $[\![\vec{p}]\!] \leftarrow \texttt{publicShare}(\vec{p})$ // Secret-share padding, known to both parties
5 $\forall i \in [n], [\![\vec{x'}]\!]_i \leftarrow [\![\vec{x}]\!]_i \,||\, [\![\vec{p}]\!]_i$ // Update data vector with padding bits
6 $[\![\vec{y'}]\!] \leftarrow \texttt{sort}([\![\vec{x'}]\!])$ // Do the ascending-order sort
7 $[\![\vec{y}]\!] \,||\, [\![\pi]\!] \leftarrow \texttt{split}([\![\vec{y'}]\!])$ // Separate sorted data from sorted padding
8 **if** *order* == *DESC* **then**
$\quad$ // Type conversion on π so we can negate it
9 $\quad [\![\pi]\!] \leftarrow \texttt{convertElementwisePerm}([\![\pi]\!], \texttt{B} \rightarrow \texttt{A})$
$\quad$ // π represents permuted padding, convert back to domain $[n]$
10 $\quad \forall i \in [n], [\![\pi]\!]_i \leftarrow -[\![\pi]\!]_i$
// Permuted padding is the inverse of the sorting permutation
11 $[\![\sigma]\!] \leftarrow \texttt{invertElementwisePerm}([\![\pi]\!])$
12 **return** $[\![\vec{y}]\!]$, $[\![\sigma]\!]$

Protocol 6: TableSort

Input : Table T with p columns $\{C_1, \ldots, C_p\}$, integer $k \leq p$, orders $o = \{o_1, \ldots, o_k\}$, $o_i \in \{\texttt{ASC}, \texttt{DESC}\}$
Output : Table T sorted on $\{C_1, \ldots, C_k\}$

1 $\sigma \leftarrow \mathbb{I}$ // Identity permutation
2 **for** i *from* k *to* 1 *descending* **do**
3 $\quad t \leftarrow C_i$ // Temp copy of column i
4 $\quad \texttt{applyPerm}(t,\ \sigma)$ // Permutation so far
5 $\quad _, \sigma' \leftarrow \texttt{VectorSort}(t,\ \texttt{o}_i)$ // Sort temp column
6 $\quad \sigma \leftarrow \texttt{composePerm}(\sigma,\ \sigma')$ // Build iteratively
// Apply final permutation to all columns
7 **for** i *from* 1 *to* p **do**
8 $\quad \texttt{applyPerm}(C_i,\ \sigma)$

5.4 Table sort

Orq sorts tables in place in Protocol 6 (TableSort) using the sorting wrapper protocol VectorSort. Each call to `sort` executes the sorting wrapper protocol, and the output is the secret-shared sorting permutation. The protocol expects a table and a subset of columns (1 to k) to use as sorting keys, along with directions to sort each sorting key. The strawman approach is to sort the entire table for each key, which would incur expensive communication under MPC. This strawman is commonly used in other MPC systems that sort multiple columns, such as Secrecy [LKFV23]. TableSort avoids this overhead as follows: it extracts sorting permutations (i.e., mappings from old to new positions) from key columns in line 5, composes them from right to left (line 6), and applies the final permutation to all columns of the table once at the end (lines 7-8).

Unlike prior works, TableSort uses different base sorting protocols depending on the key bitwidth. By default, it uses quicksort for large bitwidths (e.g., $\ell = 64$) and radixsort for small bitwidths ($\ell \leq 32$).

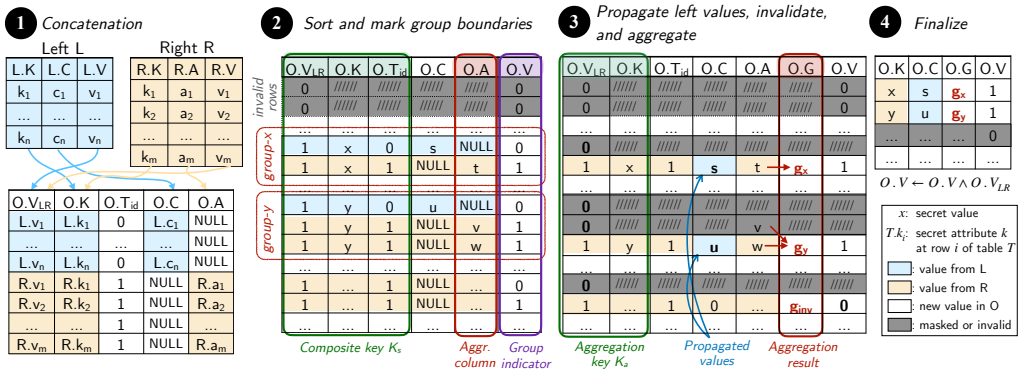

Figure 5: Steps of the basic ORQ operator. The input tables are concatenated and sorted to bring valid rows with the same join key K (e.g., `x` and `y`) next to each other. Values in $O.C$ are copied to matching rows from R. Values in $O.A$ are aggregated per key K and stored in $O.G$.

6 A composite join-aggregation operator

In this section, we introduce our approach to evaluating the composite join-aggregation operator, its extensions (§6.1–6.3), and walk through a step-by-step example of a multi-way join query (§6.4). Our new operator runs in $O(n \log n)$ time, a significant speedup over the quadratic-time Cartesian product, and, crucially, allows us to bound the size of intermediate results and chain multiple joins together to execute complex analytics.

Notation. To simplify the presentation, we avoid secret-sharing notation and describe the control flow of operators as if they are applied to plaintext data. We denote a column C in table T as $T.C$. For a row $r \in T$, we denote its C value as $r.C$. When saying *"an operator is applied to table T,"* we mean that ORQ parties perform MPC operations on the secret-shared table T, and produce secret shares of the result.

Let L, R be two tables with dimensions $n \times p$ and $m \times q$, respectively. $L.V$ and $R.V$ are the special validity columns. Without loss of generality, we assume that L and R are joined on j common columns $K = \{K_1, \ldots, K_j\}$, $0 < j \leq p, q$. We collectively refer to the columns in K (and their values) as the *join keys* $L.K$ and $R.K$. The join keys can be arbitrary columns and should not be confused with Primary Keys (PK) or Foreign Keys (FK). Our operator is more general than a PK-FK join, in that it handles cases where rows in either input may not have matching keys on the other side. It is also not equivalent to a sort-merge join, in that it computes a join and an (optional) aggregation within the same oblivious control flow (to reduce MPC costs). Instead, be build a fused join-aggregation operator based on oblivious sort and an aggregation network.

Overview of basic operator. We first describe the basic skeleton for a simple equality join that requires $L.K$ to be unique, but allows $R.K$ to contain duplicates. Later on, we discuss how to further generalize the operator to other join types, including joins with duplicate keys in both inputs. The operator joins L with R (denoted as $L \bowtie_K R$) to identify all pairs of rows from L and R with the same key K. The result is a new table O that includes all columns from R along with the specified columns C from L. Given that keys in L are unique, the cardinality of table O is at most $m = |R|$, enabling ORQ to chain multiple such join operators while keeping memory bounded. If an aggregation function is provided, the operator also applies the function to each group of rows in O

Protocol 7: JOIN-AGGREGATION (JOINAGG)

Input : Tables L, R; keys $K = \{K_1, \cdots, K_j\}$, columns $C = \{C_1, \cdots, C_k\}$ to propagate from L to R, and an aggregation function f_{agg} to apply to column A

Result : Table O

// Step 1: Initialize table O

1 $O \leftarrow \text{CONCAT}(L, R)$ // Schema shown in Fig. 5

// Sorting & Aggregation keys

2 let $K_s = \{O.V_{LR},\ O.K,\ O.T_{id}\}$, $K_a = \{O.V_{LR},\ O.K\}$

// Step 2: Sort table O on K_s (ASC)

3 $\text{TABLESORT}(O, K_s)$

4 $O.V \leftarrow O.V_{LR} \wedge \neg\text{DISTINCT}(O,\ K_a)$ // Mark groups

// Step 3: Join and aggregate per group

5 $\text{AGGNET}(O,\ \{K_a,\ C,\ f_{\text{copy}}\},$ // Copy C into R
$\{K_s,\ V,\ f_{\text{copy}}\},$ // Propagate V
$\{K_s,\ A,\ f_{\text{agg}}\})$ // Aggregate

// Step 4: Remove redundant rows and columns

6 $\text{FINALIZE}(O)$

with the same key K. Protocol 7 shows the pseudocode and Figure 5 depicts the internal steps of the operator.

Step 1: Concatenation. In the first step, the operator initializes the output table O with $n + m$ rows and $p + q - 1$ columns from the two input tables. It combines the validity columns $L.V$ and $R.V$ into $O.V_{LR}$ and the join key columns $L.K$ and $R.K$ into $O.K$. Next, it appends an auxiliary column $O.T_{id}$ and initializes rows $[1 \ldots n]$ with 0 and rows $[n + 1 \ldots n + m]$ with 1, indicating the origin table (L and R, respectively) of each row. Finally, it expands the non-key columns from L and R with n (resp. m) `NULL` values and appends them to O. Figure 5 shows a concatenation example. For simplicity, we assume one key (K) and one non-key column per table (C and A resp.). The resulting table O has $p + q - 1 = 5$ columns.

Step 2: Sort and mark group boundaries. The operator sorts table O on the composite key $K_s = \{O.V_{LR}, O.K, O.T_{id}\}$ using the table sort protocol from §5. This way, valid rows with the same key K are clustered together, and each row from L becomes the first row of its respective K group. Each group of rows in O has at most one row from L and zero or more rows from R. Next, the operator calls oblivious deduplication from §4 with key $K_a = \{O.V_{LR}, O.K\}$ to mark the first row of each group. It then flips the resulting bit and ANDs it with $O.V_{LR}$ to update the validity column $O.V$. The first row of each group now has $O.V = 0$.

Step 3: Propagate values from L, invalidate rows, and apply aggregation. In this step, the operator calls the AGGNET protocol from §4 to apply two functions: f_{copy} and f_{agg}. Function f_{copy} is an internal function that takes as input a pair of values from $O.C$ and simply returns the first one, i.e., $f_{\text{copy}}(x, y) = x$. It is used by ORQ to copy non-key values from L into R by obliviously populating the `NULL` values, and also to invalidate rows from R that did not match with any row from L. Function f_{agg} is a user-provided aggregation function. The operator applies f_{agg} to column A and stores its result in a new column $O.G$. The function f_{agg} can be one of the common aggregations from §2.1 or a custom function constructed as we explain in Appendix D.

Step 4: Finalize result. At this point, O has $n + m$ rows; however, we know that the actual output of the join contains at most m valid rows. In the final step, ORQ removes redundant columns (e.g., $O.V_{LR}$, $O.A$, $O.T_{id}$ in Figure 5) and trims all unnecessary rows which came from L. To remove these rows, we sort O on $O.V$ (in descending order), placing valid rows above invalid ones, and trim the last n rows from the sorted table.

In practice, we observe that the trimming operation is cheap but not costless, even when using our single-bit radixsort protocol: if $n \ll m$, the overhead of sorting by $O.V$ may be higher than the improvement gained in subsequent operations with the trimmed table. To address this tradeoff, we develop a heuristic that *automatically* estimates whether trimming pays off, based on the sizes of the input tables. For example, in our semi-honest honest-majority protocol, the operator trims when $9m < n \lg n \lg \ell$, where ℓ is the share representation bitwidth. Using this (conservative) heuristic has improved end-to-end execution times modestly across all queries of §8 and bounds the maximum working table size in queries with many joins. For a detailed analysis, see Appendix D.

6.1 Generalizing to different join types

JOINAGG computes inner joins with equality predicates; however, its control flow can support different join types with only minor changes to the way we invalidate rows. All operators of this section require unique keys on the left input (one-to-many) like the basic operator. We describe how to handle duplicates in both inputs in §6.3. For all join types we discuss, we provide correctness proofs in Appendix D.

Semi-join. A semi-join $L \ltimes_K R$ returns all rows in L that matched with any row from R on K. ORQ implements the semi-join by simply running JOINAGG with the two input tables swapped. After sorting O, rows from R will now precede rows from L in all groups that contain rows from both tables. The operator will therefore invalidate (and later trim) all rows from R and will only invalidate those rows from L that do not match with any row from R (i.e., they appear at the beginning of a group). In all other cases, the validity of rows from L remains as is (equal to $O.V_{LR}$), matching the semantics of semi-join without any changes in Protocol 7.

Outer joins. A left-outer join ($L \mathbin{⟕}_K R$) returns all pairs of rows from the two inputs that have the same key K plus all remaining rows from L that did not match with any row from R. To support a left-outer join, we only need to replace line 4 in JOINAGG with:

$$O.V \leftarrow O.V_{LR} \wedge \neg(O.T_{id} \wedge \textsc{Distinct}(K_a))$$

This will invalidate the first row of each group only if it comes from R. If the row comes from L, its $O.V$ is set equal to $O.V_{LR}$. This ensures that valid rows from L do not get invalidated, even if they have no matches in R, but unmatched rows from R will be invalidated as in the basic operator.

A right-outer join ($L \mathbin{⟖}_K R$) works the opposite way: it returns all pairs of rows from the two inputs that matched on K plus all remaining rows from R that did not match with any row from L. In this case, we only need to replace line 4 in Protocol 7 with $O.V \leftarrow O.V_{LR} \wedge O.T_{id}$. The line invalidates all rows from L, since they have $O.T_{id} = 0$, and leaves the validity of rows from R intact (equal to $O.V_{LR}$). A right-outer join does not need to propagate valid bits and therefore omits this step while executing the AGGNET protocol in line 5.

Finally, the full-outer join ($A \mathbin{⟗}_K B$) does not invalidate any rows from either input. In this case, the operator sets $O.V \leftarrow O.V_{LR}$ in line 4 of Protocol 7 and does not propagate any valid bits, as in the right-outer join above. All outer joins return $n + m$ rows and never trim rows from O.

Anti-join. An anti-join $L \triangleright R$ returns all rows from L that do not match any row from R. Orq implements the anti-join by executing Protocol 7 with the two input tables swapped and line 4 replaced with $O.V \leftarrow O.V_{LR} \wedge O.T_{id}$, as in right-outer join. However, here we must propagate valid bits from R to L, which is done by invoking f_{copy}. When a row exists in R, it will be sorted above the rows from L and invalidated. Then, its valid bit ($= 0$) will be copied down to all other rows, including those from L, in its group.

Theta-join. Orq's operator can support one-to-many joins with θ-predicates, as long as θ is conjunctive and contains one or more equality conditions, e.g.,

$$L.\texttt{Name} = R.\texttt{Name} \ \wedge \ L.\texttt{Time} \leq R.\texttt{Time}$$

In this case, Orq executes Protocol 7 using the equality predicate(s) that bound the output size and reduces all other predicates in θ into oblivious filters.

6.2 Supporting aggregations

In line 5 of JoinAgg, we execute an aggregation in the same oblivious control flow (AggNet) as the join. In fact, Orq can evaluate multiple aggregation functions f_{agg} with keys K_a on any combination of columns from L and R. If the query includes aggregations with keys different than K_a, an additional invocation of TableSort followed by AggNet is necessary. In case the join or aggregation keys are a prefix of the other, we can sort on the largest prefix in line 3 of JoinAgg to avoid the extra TableSort call.

Orq supports decomposable aggregation functions of the form

$$f_{\text{agg}}(X) = f_{\text{final}}(f_{\text{post}}(f_{\text{pre}}(X)))$$

where f_{pre} is a partial aggregation function over a set X, f_{post} combines partial aggregates, and f_{final} generates the final result. Self-decomposable functions are those with $f_{\text{final}} = \mathbb{I}$, the identity function. Function f_{pre} must itself be self-decomposable, $f_{\text{pre}}((X, Y)) = f_{\text{post}}(f_{\text{pre}}(X), f_{\text{pre}}(Y))$. Examples include `min`, `max`, `sum` (where $f_{\text{post}} = f_{\text{pre}}$), and `count` (where $f_{\text{post}} =$ `sum`). Note that self-decomposability implies commutativity and associativity [JBA14], which are necessary for AggNet correctness. Decomposable aggregations enable efficient incremental computation and data-parallelism, and have been studied extensively in the context of sensor networks [MFHH05] and dataflow systems [YGI09]. In Orq, we employ similar ideas in oblivious computation to bound the size of intermediate results, as we explain next.

6.3 Supporting joins on duplicate keys

We can use JoinAgg to handle many-to-many joins (with duplicate keys on both sides), as long as the join is followed by a decomposable aggregation and the aggregation keys belong to one input table (see §2.1). This ensures that the worst-case cardinality of the output table is equal to the cardinality of the table with the aggregation key(s).

Decomposable functions are amenable to partial evaluation; therefore, we can apply them without having to materialize the Cartesian product of the two input tables. As such, we apply f_{pre} before the join, f_{post} after the join, and, optionally, f_{final} to compute the final result. Let K_j and K_a be the join and aggregation keys, respectively. Orq applies function f_{pre} to one of the input tables to compute a partial aggregation on K_j. The result is a table with unique join keys that can now be joined with the second table using the basic operator. Function f_{post} is evaluated on the output table O with aggregation key K_a.

As an example, consider a join $L \bowtie_{K_j} R$ on K_j with duplicate keys in both input tables followed by $f_{\text{agg}} =$ `count` that outputs the number of rows in the join result. We can

obliviously compute the aggregation without a quadratic blowup by first evaluating function $f_{\text{pre}} = \texttt{count}$ on L to count the number of rows per key K_j (multiplicity). Then, we can join L with R using JoinAgg and apply $f_{\text{post}} = \texttt{sum}$ to add the multiplicities. The idea also applies to acyclic queries with multi-way joins like in plaintext query evaluation [JPR16].

6.4 Putting it all together

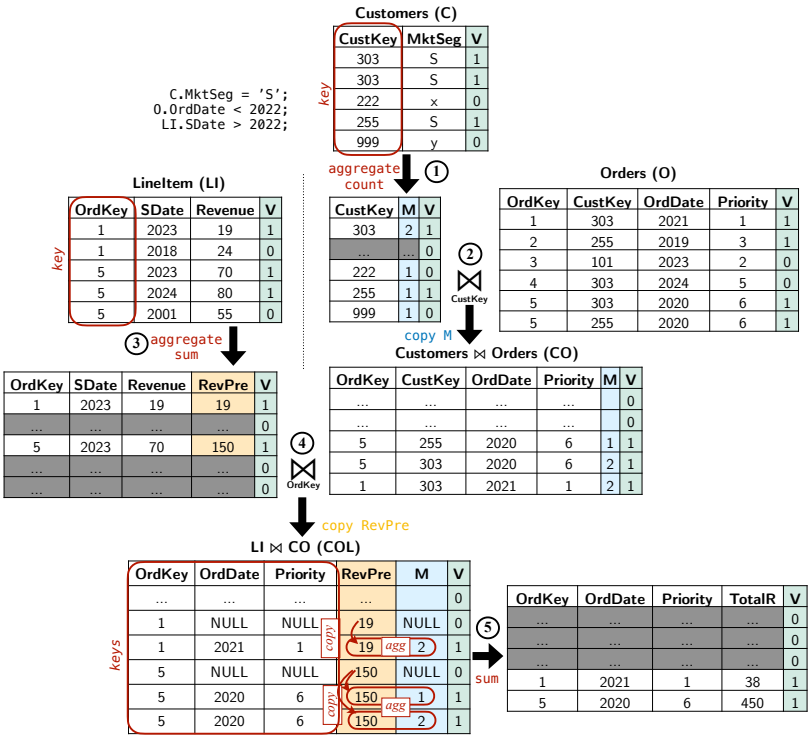

Figure 6: Step-by-step oblivious evaluation of TPC-H Q3 assuming no PK-FK constraints (all input tables have duplicate join keys).

Figure 6 shows an example with many-to-many joins, using a generalization of Q3 (see Listing 1), where we allow duplicates in all columns. That is, we assume no public knowledge of the PK-FK constraints in the TPC-H specification of this query. We omit filter operations to keep the example simple.

To evaluate $C \bowtie_{\texttt{CustKey}} O$, we first apply a pre-aggregation to C that computes the multiplicity of each `CustKey` using Protocol 2. The pre-aggregation makes `CustKey` in C unique and stores multiplicities in a new column `M`. Next, we join C with O on `CustKey` using Protocol 7 to produce a new table CO. The second join $LI \bowtie_{\texttt{OrdKey}} CO$ is evaluated similarly: we first apply a pre-aggregation to LI that computes the `Revenue` per `OrdKey` and stores it in column `RevPre`. Then, we compute the join using Protocol 7 and apply the post-aggregation to sum the product `RevPre`*`M` per key $K_a = \{\texttt{OrdKey}, \texttt{OrdDate}, \texttt{Priority}\}$ using Protocol 2. Listing 2 shows the implementation using the Orq API.

7 System implementation

We have implemented Orq from scratch in C++. Figure 7 shows the software stack run by each computing party, which includes (i) an MPC layer with vectorized implementa-

```
// Pre-aggregate and apply first join
auto CO = C.aggregate (   {"CustKey"}, {"M", "M", count})
           .inner_join(O, {"CustKey"}, {"M", "M", copy});
// Pre-aggregate and apply second join
auto COL = LI.aggregate (
                    {"OrdKey"}, {"Revenue", "RevPre", sum}
            ).inner_join(
                CO, {"OrdKey"}, {"RevPre",  "RevPre", copy});
// Apply post-aggregation
COL["TotalR"] = COL["RevPre"] * COL["M"];
auto RES = COL.aggregate(
                    {"OrdKey", "OrdDate", "Priority"},
                    {"TotalR", "TotalR", sum});
```

Listing 2: The example of Figure 6 implemented in ORQ

tions of secure primitives, (ii) a library of oblivious relational operators, (iii) a custom communication layer, and (iv) a data-parallel execution engine. Users can configure the share size 2^ℓ ($\ell = 64$ by default), the MPC protocol, and the sorting protocol (quicksort by default), according to their application needs.

We use `libOTe` [RR24] to produce random oblivious transfers, from which we generate OLE (Oblivious Linear Evaluation) correlations and Beaver triples, as described in §3.4.1. For permutation correlations, we use the technique by Peceny et al. [PRRS24]. For random number generation, we use `AES` from `libsodium` [lC25].

Protocol-agnostic secure operator abstractions. All relational operators, sorting, multiplexing, and division with private denominator are protocol-agnostic. The protocol-specific primitives are the functionalities defined by each MPC protocol (e.g., $\times$, $\wedge$, etc.), division by public denominator, and two shuffling primitives that we use to generate correlated randomness (random sharded permutations). Higher-level shuffle operators, such as composing, inverting, converting, and applying permutations, are protocol agnostic, and make use of these protocol-specific correlations. For more details, see Appendix A.

Vectorization and data parallelism. ORQ uses a columnar data format that allows for efficient vectorized execution. Recall that each secure multiplication and bitwise AND operation incurs a message exchange between parties. With vectorization, ORQ can efficiently group independent messages within operations and exchange them in the same network call. All low-level primitives and oblivious operators in ORQ are vectorized to work this way.

Each ORQ server runs as a single process with the same (configurable) number of threads τ. Execution within a server is data-parallel: a coordinator thread assigns tasks to workers, which operate on disjoint partitions of the input vector. ORQ's communication layer establishes a static mapping between workers, so that thread with id t communicates only with thread t in other servers, $0 \leq t < \tau$.

Communication layer. In ORQ, we implement a TCP-based communicator, tailored to I/O-intensive MPC execution. The communicator manages network sockets via a configurable number of dedicated threads. A high number of communication threads can significantly improve performance in WAN settings, where multiple parallel connections between parties can be used to leverage available bandwidth.

Worker threads interact with communication threads using lock-free ring buffers to avoid blocking computation. Communication threads are decoupled from worker threads

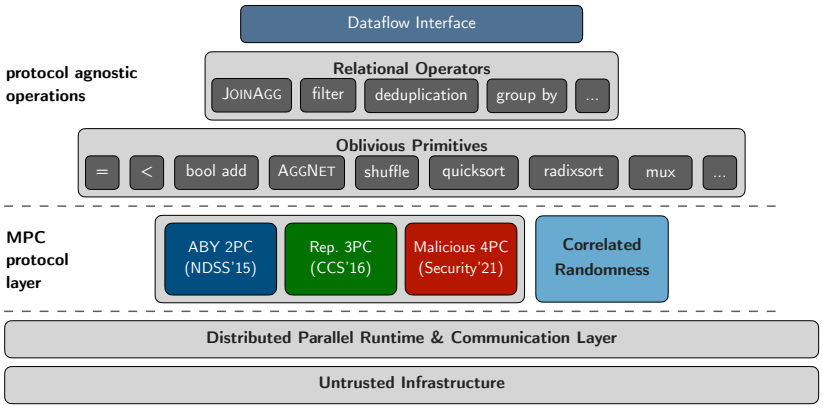

Figure 7: The ORQ stack. Analysts interact with the framework through its C++ dataflow interface. All oblivious primitives and relational operators inherit the security guarantees of the underlying MPC protocols.

and process ring buffers in a round-robin fashion. Data copies are avoided both ways; worker threads pass offsets in vectors they operate on, and the communicator directly pulls from (on send) and pushes to (on receive) these vectors.

8 Experimental evaluation

Our experimental evaluation is structured into five parts:

ORQ's performance on complex analytics. In §8.2, we present a comprehensive performance evaluation of ORQ in LAN and WAN with three MPC protocols. Our results demonstrate that ORQ provides excellent performance and can compute complex multi-way join queries on millions of input rows within minutes.

Comparison with state-of-the-art. In §8.3, we compare ORQ with the following state-of-the-art systems:

(i) Secrecy [LKFV23], a relational MPC framework targeting the 3-party outsourced setting with semi-honest security and no leakage. Secrecy is the only open-source relational system for this setting.

(ii) SecretFlow [FCH+24], a relational MPC framework targeting the 2-party peer-to-peer setting with semi-honest security. SecretFlow is the only system we are aware of that scales some TPC-H queries to millions of rows per table, albeit by leaking information to parties.

(iii) MP-SPDZ [Kel20], a general-purpose MPC compiler used in many prior works. While MP-SPDZ does not support relational analytics, we compare with its sorting implementation, as sorting is the most expensive operator in relational queries.

ORQ provides up to 827× lower query latency than Secrecy and its secure sorting implementations are up to 5.9× and 189.1× faster than the ones provided by SecretFlow and MP-SPDZ, respectively. We also achieve modest speedups over SecretFlow on queries with joins (up to 1.5×), despite the fact that SecretFlow introduces leakage while ORQ does not.

Scalability evaluation. In §8.4, we show that ORQ's operators scale to hundreds of millions of rows under all protocols. When configured with our most expensive protocol for malicious security, ORQ sorts 2^{29} rows in less than 2 hours. We also report results on ORQ's performance in a geodistributed setting in §8.5, where we show that ORQ can be practically deployed over the Internet.

Ablation study. In §8.6, we consider each of our major optimizations separately and measure how they affect end-to-end runtime on a representative query.

Preprocessing. Finally, we benchmark protocols for generating boolean and arithmetic Beaver triples in §8.7.

8.1 Evaluation setup

We use `c7a.16xlarge` AWS instances with Ubuntu `22.04.5 LTS` and `gcc 11.4.0`, in two configurations: (i) `LAN` is an unconstrained setting in the `us-east-2` (Ohio) region, with a maximum bandwidth of 25Gbps and a 0.3ms round-trip time (RTT), (ii) `WAN` is a restricted setting in `us-east-2`, with symmetric RTT of 20ms and 6Gbps bandwidth (as measured between `us-west-2` and `us-east-1` with 16 parallel `iperf3` streams). Unless otherwise specified, ORQ is configured with 16 compute threads and 4 parallel connections in `LAN` and 16 connections in `WAN`.

Workloads. We use 31 queries for evaluation. Our most comprehensive benchmark is the full set of TPC-H queries [Tra24], implemented with the ORQ dataflow API. We replace floating point values with equivalent integral values and pattern matching expressions (`X like 'Y%'`) in six queries with oblivious (in)equality. To perform fully private averages, we implement a *non-restoring division* circuit inspired by the hardware literature [Lu05]. Various prior works have implemented selected TPC-H queries [LKFV23, PKY+21, BHE+18, FCH+24], but to our knowledge, we are the first to report performance numbers for *all* TPC-H queries (22 in total) entirely under MPC. We also implement all other queries we could find in prior relational MPC works [VSG+19, LKFV23, PKY+21, BHE+18, FCH+24, WY21, BEE+17, LWDY24, BPH+20]:

- *Aspirin*, *C. Diff*, *Password*, *Credit Score*, *Comorbidity*, and *SecQ2*, used in Secrecy [LKFV23], Conclave [VSG+19], and Senate [PKY+21], among others.
- *Market Share* from Conclave.
- *SYan*, Example 1.1 from Wang et al. [WY21].
- *Patients*, a 3-way join query used in Shrinkwrap [BHE+18] to showcase the cascading effect (which ORQ avoids).

Additionally, we implement the five peer-to-peer variations of TPC-H queries in SecretFlow [FCH+24], denoted as *S1–S5*. As an extra correctness check, we also implemented all queries in SQLite [SQL25] and compared its output with ORQ's.

Inputs. In all experiments, we use $\ell = 64$ bit shares by default. However, because our sorting protocols pad inputs to guarantee uniqueness and compute permutations, sorting is actually performed over 128-bit shares. We standardize a notion of query size: in the TPC-H specification [Tra24], input tables grow according to the *scale factor* (SF) parameter, where SF1 corresponds to an (approximately) 1GB database. The smallest table (`Supplier`) has $10\text{k} \cdot \text{SF}$ rows, while the largest (`LineItem`) has $6\text{M} \cdot \text{SF}$ rows. TPC-H queries have total input sizes between $810\text{k} \cdot \text{SF}$ (Q11) to $8.5\text{M} \cdot \text{SF}$ (Q9) rows, and average

5.8M · SF rows. For non-TPC-H queries, we accordingly define input sizes with ≈ 5M rows per SF.

Protocols. We label results according to the MPC protocol: **SH-DM** (semi-honest, dishonest majority) for ABY [DSZ15], **SH-HM** (semi-honest, honest majority) for Araki et al. [AFL+16], and **Mal-HM** (malicious-secure, honest majority) for Fantastic Four [DEK21]. In all *SH-DM* experiments, we report online time, as in prior works [FCH+24]. A recent attack [BS26] was published on the security guarantees of the Fantastic Four protocol. The authors notified us of their attack while this paper was under submission; we include notes below on how the fixed, secure implementation affects our results. Notably, Brüggemann et al. [BS26] attack the protocol itself, as opposed to any particular implementation, so MP-SPDZ [Kel20], which we compare to, was likewise vulnerable. For an in-depth discussion of how we patched our implementation, see our technical report [Bau26].

8.2 Performance on complex analytics

In this experiment, we run Orq at Scale Factor 1 (SF1) on the full workload, which includes queries of varying complexity. Queries like Q5, Q7, Q8, Q9 and Q21 are expensive, involving 4-7 joins each, with Q21 also performing a self-join on the largest input table, `LineItem`. Other queries include multiple filters, semi-joins, outer joins, group-by aggregations, distinct, and order-by operations, covering a wide range of query patterns. Q6 is the least expensive query, as it does not require sorting. We observe than the fixed version of Mal-HM [Bau26] has an overhead of ≈ 2× on some queries at Scale Factor 1, primarily due to the additional malicious checks in sorting, but is up to ≈ 3× faster in other cases, due to the better resource utilization of batched malicious checks. Our results here correspond to the `custom` protocol in that paper.

Figure 8 shows the end-to-end times in `LAN` and `WAN` and summarizes the result statistics. Overall, Orq computes complex multi-way join queries on millions of input rows in a few minutes. The most expensive query, Q21, calls the sorting operator 12 times and, under malicious security, completes in 42 minutes over `LAN`. In the same setting, Orq executes all other queries from prior work in under 10 minutes. Our results also demonstrate the effectiveness of Orq's vectorization and message batching. In the `WAN` environment, we see 1.2×-6.9× higher execution times for a 75× higher RTT.

8.3 Comparison with state-of-the-art systems

In this section, we compare Orq with three state-of-the-art MPC frameworks on relational queries and oblivious sort.

Comparison on queries. We first compare Orq with Secrecy [LKFV23], the only open-source relational MPC system without leakage that operates in the outsourced setting. We configure Orq with the SH-HM protocol, which is the one used by Secrecy, and we execute all queries from the Secrecy paper, using the maximum input size they report [LKFV23, Fig. 4].

Figure 9 (left) shows the results. For the most expensive queries (Aspirin, Q4, and Q13) that perform a join or semi-join, Orq achieves 478× — 760× lower latency. Password, Credit, Comorbidity, and C.Diff include group-by and deduplication operators. For these queries, Orq's optimized sorting protocols also improve latency by 17× — 42×. Q6 does not have any join or sorting, yet Orq outperforms Secrecy by 3×. The significant improvements come from better asymptotic complexity of our join and sorting algorithms. Secrecy uses an $O(n^2)$ join and an $O(n \log^2 n)$ bitonic sort, while Orq evaluates join-aggregation queries in $O(n \log n)$. Also, Secrecy's runtime is single-threaded.

	TPC-H		Other	
	Median	Max	Median	Max
SH-DM LAN	3.8	15.0	1.6	3.1
WAN	13.5	52.7	6.2	11.7
SH-HM LAN	4.4	17.4	2.0	3.7
WAN	10.9	41.4	4.8	9.1
Mal-HM LAN	10.9	42.3	4.9	8.3
WAN	27.1	108.4	11.9	21.7

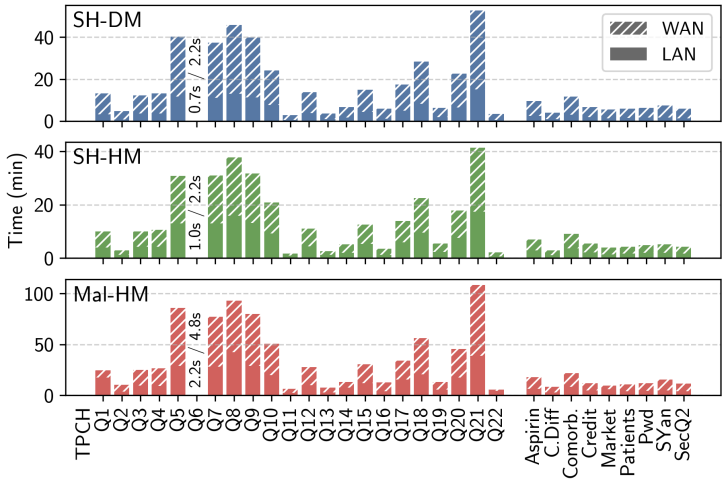

Figure 8: Query execution time (min) at SF1 in `LAN` (solid) and `WAN` (hatched). Bars are overlapped, not stacked. TPC-H queries shown on the left and other queries on the right. Q6 times are annotated.

Next, we compare with SecretFlow [FCH+24], a recent relational MPC framework that operates in the peer-to-peer setting. To our knowledge, SecretFlow is the only open-source system of this type that scales TPC-H-based queries to millions of input rows per table. However, we emphasize that SecretFlow *leaks which rows match* to the parties, while Orq does not introduce any leakage. Nevertheless, we include this comparison to show that Orq can achieve competitive performance even in a peer-to-peer setting. We configure Orq with the SH-DM protocol that uses the same underlying MPC primitives as SecretFlow (ABY [DSZ15]), and we allow both systems to perform operations in the data owners' trusted compute, otherwise SecretFlow's optimizations are not applicable.

Figure 9 (right) shows results for the five queries used in the SecretFlow paper, run on 16M input rows per table. Orq delivers superior performance in all cases, while protecting all data and intermediate result sizes. It achieves $58 - 85\times$ lower latency for simple queries without joins (S1, S2) and $1.1 - 1.5\times$ lower latency for join queries that include aggregations (S3, S4) or oblivious group-by (S5). S1 and S2 see such significant speedups because — under SecretFlow's peer-to-peer model — those queries largely reduce to local SQL operations rather than communication-intensive MPC evaluations. Despite SecretFlow's leakage, Orq still wins on performance due to its more efficient sorting and fused join-aggregation operator. Both systems use radixsort, but SecretFlow cannot leverage parallelism. In contrast, Orq's operators are vectorized and data-parallel.

Comparison on oblivious sort. We now compare Orq's oblivious radixsort against two publicly available state-of-the-art implementations. In Figure 10, we compare our SH-DM radixsort with SecretFlow's `SBK` (64-bit radixsort) and `SBK_valid` protocols. `SBK_valid` is a modified radixsort that relies on data owners locally mapping plaintext data to use at most $\lceil \lg n \rceil$ bits, where n is the input size ($n = 17, 20, 24$ bits in this experiment). Even

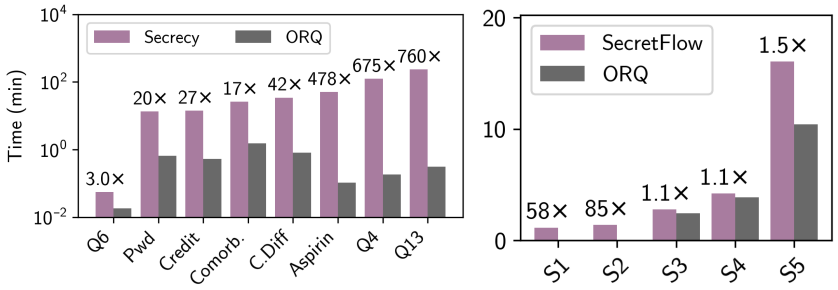

Figure 9: ORQ query times compared to Secrecy and SecretFlow. Secrecy operates in the outsourced setting without leakage. SecretFlow is a peer-to-peer system that offloads operations to the data owners' trusted compute and leaks the result of the join to parties.

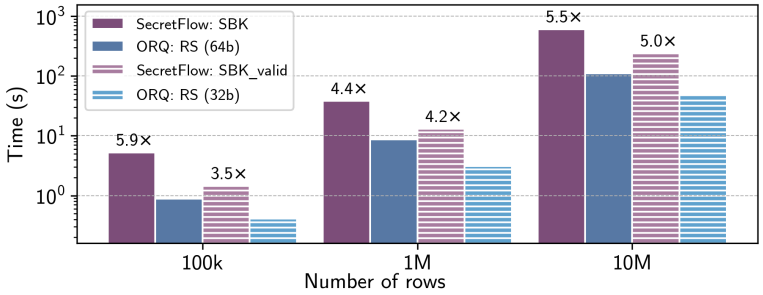

Figure 10: Performance of oblivious sort in SecretFlow and ORQ.

though it favors the baseline, we compare `SBK_valid` to our 32-bit radixsort and `SBK` to our 64-bit protocol. We see that ORQ is up to 4.4× and 5.5× faster when sorting 1M and 10M elements, respectively.

Our next comparison is with MP-SPDZ [Kel20], a state-of-the-art general-purpose MPC library that provides secure sorting implementations for all MPC protocols supported by ORQ. We compare with its radixsort implementation, which is its most efficient sorting algorithm. Figure 11 shows the results. We vary the input size by powers of two, starting from 2^{16} elements and increasing the input size until MP-SPDZ either runs out of memory or crashes: up to 2^{22} elements with SH-DM, 2^{25} with SH-DM, and 2^{20} with Mal-HM. In the SH-DM and Mal-HM settings, ORQ's radixsort is 189× and 134× faster than the corresponding MP-SPDZ implementations. Under the SH-HM protocol, which is heavily optimized in MP-SPDZ, ORQ achieves a 8.5× speedup when sorting 2^{24} elements, however, MP-SPDZ could not complete the experiment for larger inputs. The large performance improvements over MP-SPDZ come from data-parallelism; although MP-SPDZ supports parallelism and advanced vectorization, it does not parallelize sorting. Due to the attack by Brüggemann et al. [BS26], our Mal-HM results are expected to be slightly slower. However, we note that the implementation of Fantastic Four in MP-SPDZ was *also* vulnerable to the same attack, so is expected to pay a similar, if not greater, cost. All experiments were run before the attack was made public. With our initial fixed implementation [Bau26], we observe a sorting overhead of about 3-6× in this regime, which quickly amortizes with larger input sizes.

Bandwidth consumption. We also evaluate bandwidth consumption in ORQ and the baselines. When compared to Secrecy, ORQ exhibits up to two orders of magnitude lower bandwidth consumption, due to our asymptotic improvements in sorting and join. Likewise,

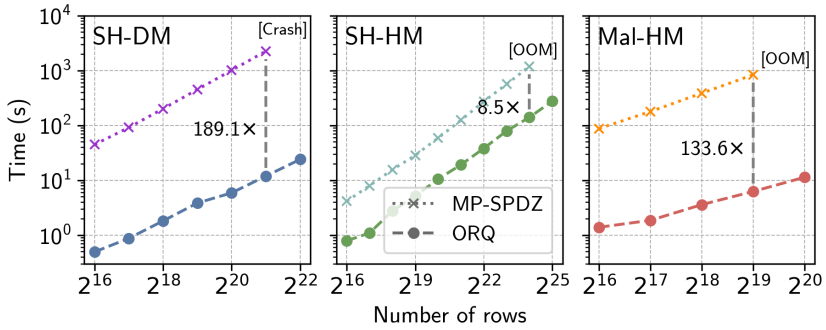

Figure 11: Performance of oblivious radixsort protocols in MP-SPDZ and Orq. We run MP-SPDZ until it crashes or runs out of memory.

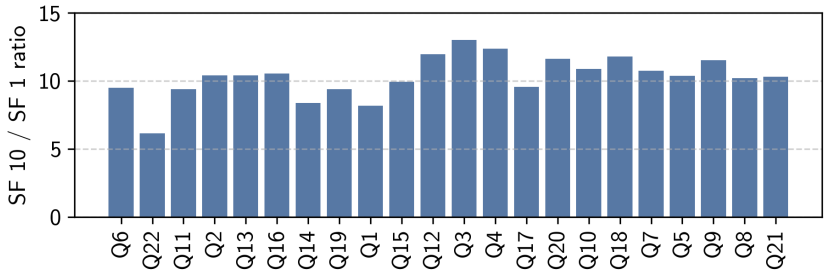

Figure 12: Ratio of TPC-H query execution times at SF10 over SF1 for the SH-DM protocol in `LAN`. The 22 queries are sorted from left to right by increasing SF10 latency.

our sorting implementations achieve 1.6× lower bandwidth than MP-SPDZ SH-HM and more than an order of magnitude improvement against MP-SPDZ SH-DM and Mal-HM. Orq also achieves a 1.8× lower bandwidth than SecretFlow in sorting. On queries, SecretFlow has a lower bandwidth than Orq, as it leaks matching rows in its join operator and performs subsequent operations locally. We report detailed bandwidth measurements for all experiments in Appendix E.

8.4 Orq scalability

In this section, we show that Orq can deliver practical performance for complex analytics on large inputs, without sacrificing security. To the best of our knowledge, we are the first to report results at this scale, for end-to-end MPC execution without leakage or the use of trusted parties.

TPC-H at SF10. We repeat the experiment of §8.2 on the TPC-H benchmark at Scale Factor 10. In this setting, queries operate on inputs of 58M rows, on average. The most expensive query, Q21, has a total input of size 75M rows and performs two of the largest joins, sorting tables with 120M rows each time.

Figure 12 plots the ratio of execution times at SF10 over SF1 for the SH-DM protocol in `LAN`. Assuming that query latency is dominated by sorting and aggregation, we would expect the runtime to scale as $O(n \log n)$. If the queries consisted *only* of ideal sorts, then theoretical scaling would be $10 \log(10n)/\log n \approx 11.5$, when $n \approx 5.8$M (SF1). Overall, this is indeed the trend we see in Figure 12, with some outliers. In practice, queries are more complex and consist of operations with different costs. Some overheads are sublinear: for example, oblivious division is round-constrained for most inputs, so scaling appears lower

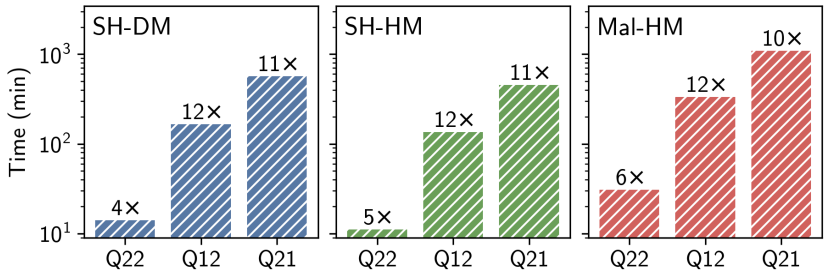

Figure 13: Orq performance at SF10 in `WAN`. Bars are labeled with the scaling ratio compared to SF1 in the same environment. Q21 is the most expensive query in the TPC-H benchmark across protocols.

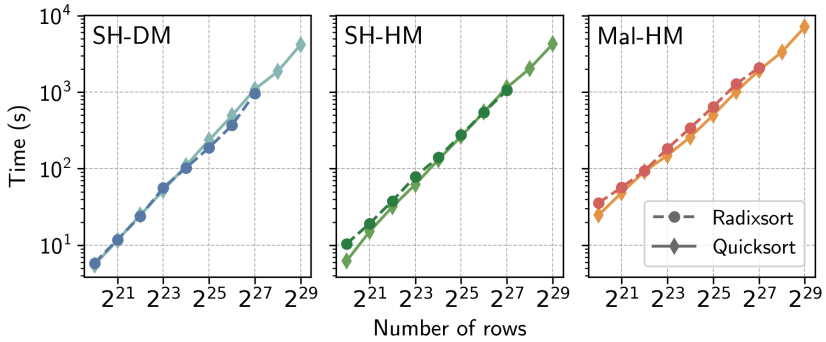

Figure 14: Scaling Orq's oblivious sorting protocols to very large inputs. Quicksort scales to larger inputs than radixsort, which has higher time and memory requirements for permutation generation.

for queries with division, like Q22. Other queries suffer slightly suboptimal scaling, due to the power-of-two padding required in AggNet. For example, Q12 aggregates over a table of size 7.5M at SF1, which is padded to $2^{23} \approx 8$M rows. However, at SF10, this same aggregation occurs on a table of size *75M*, which now must be padded to $2^{27} \approx 134$M rows, 16× larger than the table at SF1.

Next, we evaluate Orq's scalability in `WAN`, using the two outlier queries mentioned above, Q22 and Q12, and the most expensive query, Q21. Figure 13 shows the results for all protocols, where each bar is annotated with the scaling ratio compared to SF1. Orq's scaling behavior in this environment is consistent with our previous findings. Under malicious security, Q22 completes in 31 minutes and Q21 in 18 hours.

Scalability of sorting protocols. In the previous experiment, we showed that Orq provides practical performance on complex queries that repeatedly sort tables with over 100M rows. We now stress Orq's oblivious sorting protocols further and evaluate their performance on even larger inputs. We run radixsort and quicksort in `LAN` with 32 threads, increasing the input size from 2^{20} to 2^{29} elements. Figure 14 shows the results. The slowest protocol, Mal-HM radixsort, can sort 134 million ($n = 2^{27}$) elements in about 35 minutes, while the fastest protocol, SH-HM quicksort, sorts 537 million ($n = 2^{29}$) in just over 70 minutes. For Mal-HM, the results on large inputs do not noticeably change after fixing the attack by Brüggemann et al. [BS26], since the fully-secure checks only incur a constant-communication overhead $O(\log n)$ times. Concretely, we observe an overhead on Quicksort of less than 1% at $n = 2^{28}$ rows [Bau26].

We find that quicksort and radixsort are competitive across settings. Quicksort is

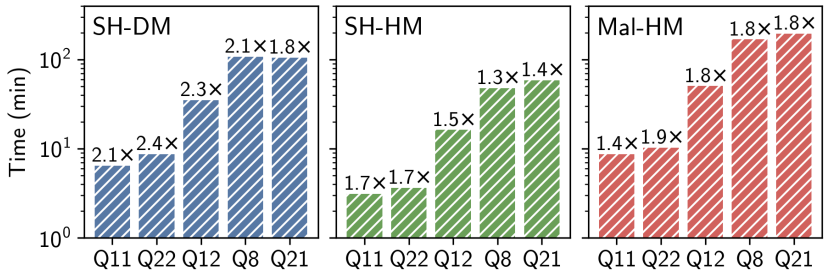

Figure 15: TPC-H query times at SF1 in a geo-distributed WAN deployment, with ratios over times in our symmetric WAN.

consistently faster under the malicious protocol, while radixsort is slightly faster in the SH-DM setting. Overall, quicksort (Orq's default sorting protocol) scales to larger inputs in our experimental setting, as radixsort has higher memory demands: it requires computing and storing $\ell + 2$ secret-shared permutations of length n.

8.5 Performance in a geodistributed setting

In this section, we evaluate Orq's ability to amortize network costs in a geodistibuted setting. We create a deployment spanning four AWS regions: `us-east-1` (N. Virginia), `us-east-2` (Ohio), `us-west-1` (N. California), and `us-west-2` (Oregon). The deployment is constrained by minimum link bandwidth between $4.23 - 8.47$ Gbps, and RTT between $50 - 61$ ms. To make the setting challenging, we deploy parties so that at least one link exists between the U.S. east and west coasts.

We show results for five TPC-H queries: two of the fastest (Q11, Q21), the median (Q12), and the two slowest (Q8, Q21). These five queries account for approximately 30% of the total time to run the entire TPC-H benchmark. Figure 15 shows the execution time, along with the overhead compared to the symmetric `WAN` environment used in the experiments of Fig. 8. Latency increases by up to 2.4× for SH-DM, 1.7× for SH-HM, and 1.9× for Mal-HM. These results demonstrate that Orq's overhead is lower than the corresponding 3× increase in RTT between the two WAN environments, indicating its ability to effectively amortize network costs and achieve competitive performance. The current implementation of the fixed malicious checks [BS26, Bau26] are not round-optimized, so may increase WAN times slightly. With further engineering effort, however, we can lower the round complexity; our analysis shows that only four rounds are necessary to run each malicious-secure check.

8.6 Ablation study

We perform ablation studies in both LAN and WAN to measure the effectiveness of various optimizations, one at a time. Our target workload is TPC-H Q15, which had a median runtime and includes all of our major operators. We focus on the SH-HM protocol and benchmark with 600k rows (SF0.1) on a cluster of 96-core AMD EPYC 9655P CPUs with 256 GB of memory each. We choose SF0.1 because the baseline does not scale beyond that. At this size, the quadratic join must materialize a table with 600M rows; at SF1, this jumps to 60B rows, well beyond the memory capacity of our machines.

As part of the ablation study, we implement the pairwise sorting network [Par92], which may be of independent interest. Like bitonic sort, it has $O(n \log^2 n)$ asymptotic complexity, but it uses $O(n \log n)$ fewer comparisons — the minimal number for any practical sorting network (and the same as Batcher's odd-even mergesort [Bat68]).

Table 1: Ablation study optimizations in LAN

Baseline	Optimized	Speedup
Sorting Network	Quicksort	11.4×
Cartesian Product	JoinAgg	58.2×
No Batching	Multiple Batches	1481.6×
Single Threaded	Multithreaded	1851.0×

Table 2: Ablation study optimizations in WAN

Baseline	Optimized	Speedup
Single Thread	Multithreaded	1.2×
Multiple Batches	Single Batch	1.7×
Single Connection	Multiple Connections	6.8×

The baseline configuration uses no multithreading, no message batching, and asymptotically slower algorithms for join and sort (both implemented in Orq): a Cartesian product with cost $O(n^2)$ and a pairwise sorting network [Par92] with cost $O(n \log^2 n)$, as used in prior work [LKFV23, HZF+22]. Tables 1 and 2 summarize the optimizations and the resulting speedups in LAN and WAN, while Figure 16 plots the end-to-end execution times.

We do not evaluate the baseline join and sorting algorithms in WAN, as the network overhead of the Cartesian product and sorting network dwarfs the system optimizations that we wish to highlight. We also omit multiple connections in the LAN test, since a single connection is able to take full advantage of available network resources in that setting. We evaluate the effect of message batching in both LAN and WAN: in LAN, we compare no message batching at all with twelve batches per thread (our default), whereas for WAN, we compare twelve versus one batch per thread, to demonstrate how the decreased round complexity improves performance in that setting (batches execute sequentially per thread, so fewer batches corresponds with lower round complexity). We do not test without batching in WAN, as the increased network latency makes the experiment impractical to complete.

In LAN, we see that each of our asymptotic improvements (sorting and join algorithm) give about a 10× and 50× speedup, respectively. In both WAN and LAN, system optimizations (threading, multiple connections, and batching) also give sizeable speedups. In WAN, the greatest speedup comes from adding multiple parallel connections, since in this setting MPC execution is bound by communication, not computation. While multithreading is a powerful tool to improve the end-to-end running times of Orq queries, it only becomes beneficial at much larger inputs than SF0.1.

Overall, we observe about 1800× higher query throughput due to our optimizations. Since our improvements are *asymptotic* — from $O(n^2)$ to $O(n \log n)$ — this performance gap will only grow with larger input sizes. We observe similar speedups in the other two protocols supported by Orq (SH-DM, Mal-HM).

8.7 Preprocessing phase

Orq uses both boolean and arithmetic Beaver triples. Our final experiment is an independent benchmark of the preprocessing phase for the SH-DM protocol, with a particular focus on the challenging case of generating arithmetic Beaver triples (over the integers mod 2^ℓ). Such triples are needed for executing the secure multiplication protocol and the arithmetic multiplexer (§4). We are concerned with the *runtime* and *communication overhead* of generating Beaver triples. We require one Beaver triple per multiplication gate;

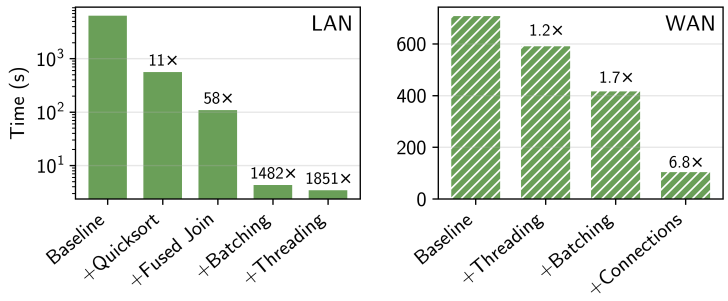

Figure 16: Ablation study results in LAN and WAN. Above each bar we label the relative speedup compared to the baseline.

a vector of length n requires n Beaver triples. For comparison purposes, we also include measurements for boolean (AND) triples. As described in §3.4.1, we construct Beaver triples via a local reduction to twice as many oblivious linear evaluations (OLE). For all experiments of this section, we use the same hardware setup as in §8.6.

One might reasonably ask why we even bother with arithmetic triples, or, for that matter, arithmetic secret sharing in the first place. In this work, we optimize for a fast *online* phase; our goal is to evaluate queries with as little latency as possible, but allow for (possibly expensive) preprocessing which may occur before input data — or even the query itself — has arrived. Without arithmetic triples, all multiplications and additions would have to be decomposed down to their expensive circuit representations (which require multiple rounds and have $O(\ell^2)$ and $O(\ell)$ communication complexity, respectively), leading to orders-of-magnitude slowdowns in the online phase.

We evaluate Gilboa's original OLE protocol [Gil99], which has communication cost $O(\ell^2)$ to generate an ℓ-bit Beaver triple. We also implement the recent landmark protocol by Doerner et al. [DHIM25] which uses a number-theoretic construction to break the two-decade-old quadratic barrier, and achieves $O(\ell \log \ell)$ communication per triple. To our knowledge, we are the first to publish an implementation for this protocol. Our results confirm that despite the heavier computational overhead of the subquadratic protocol, it is concretely more communication efficient at practical input sizes. In real-world deployments, such as wide-area networks, where bandwidth may be more expensive than compute, it is a viable alternative for arithmetic triple generation.

We base triple generation on the pseudorandom correlation generator (PCG) implementation (`SilentOTExt`) from `libOTe` [RR24], in order to take advantage of sublinearities in the number of triples generated. AND triples directly follow from oblivious transfer, so those figures represent an effective benchmark of OT generation. However, we acknowledge that more engineering work is needed to make silent OT generators fully practical. Due to memory constraints, we are often unable to generate enough triples at once to fully amortize the computational overheads of the silent constructions.

In this experiment, we benchmark triple generation with up to 64 threads, and in batches of up to 2^{24} elements. We report *normalized* throughput in Table 3, which shows the results for the fastest batch size, and Figure 17, which plots the fastest amortized execution time per bit. To enable a clearer comparison between protocols and bitwidths, we express throughput in terms of normalized megabits per second, and execution time in terms of normalized nanoseconds per generated bit. These values represent the best performance achievable with our implementation. The actual number of triples generated, in each case, is a factor of ℓ lower.

Gilboa's protocol achieves a maximum throughput of 165 Mbps at a width of 32 bits, corresponding to about 5M triples per second. The highest throughput observed for the CRT protocol was almost three times lower: 56 Mbps at 128 bits, or just over 400k triples

Table 3: Maximum throughput observed for triple generation. Measurements were taken at 64 threads. Batch sizes are the closest powers of two.

Triple Type	Bitwidth ℓ	Max. Tput	Opt. Batch Size
Boolean	8	87.4 Mbps	524k
`SilentOT` [RR24]	16	130.2	262k
	32	123.8	131k
	64	119.7	65k
	128	123.4	32k
Arithmetic	8	144.4 Mbps	4M
Gilboa [Gil99]	16	163.6	8M
	32	165.1	4M
	64	154.0	8M
	128	82.7	16M
Arithmetic	8	11.0 Mbps	4M
Doerner et al. (CRT) [DHIM25]	16	25.1	8M
	32	37.1	8M
	64	48.0	8M
	128	56.0	8M

per second. However, at this bitwidth, Gilboa only achieves 650k triples per second.

Figure 18 shows how many bits of communication are required per output bit, for both 64-bit triples (our default bitwidth) and 128-bit triples (our sorting bitwidth; see §5). In this experiment, we see the real benefit of the CRT protocol: it uses less than half the bandwidth of Gilboa's protocol at 64 bits, and 5× lower bandwidth at 128 bits. The amortized overhead of each Boolean triple in our implementation is around two communicated bits per output bit. We omit these values from Figure 18 since they would not be clearly visible, but remark that this implies approximately 26× greater communication overhead, per bit, for the CRT protocol compared to Boolean triples, and 128× greater overhead for Gilboa, in the case of 128-bit outputs. With further work on batched silent preprocessing, we could, in principle, bring the communication cost of a Boolean triple close to zero.

The CRT protocol trades off substantially lower communication for greater computation. While we do not see lower end-to-end latencies, in practice, due to greater computational overheads, our results show that, with a more efficient use of computation resources, we could achieve a concrete improvement over Gilboa's protocol. This is an exciting avenue for future research, but will require new approaches to meeting the storage demands of correlated randomness.

9 Related Work

There is a large body of work on systems for secure computation. In this section, we only focus on works that are directly related to Orq.

Relational MPC Most systems and protocols for secure relational analytics target *peer-to-peer* settings and propose techniques tailored to a small, fixed number of semi-honest data owners, who also act as computing parties: either two [FCH+24, WY21, PHP+24, BEE+17, BHE+18, BPH+20] or three [VSG+19, LWDY24]. In such settings, performance can be improved by splitting the query into (i) a plaintext part that data owners compute locally (e.g., filters, hashing, sorting, and grouping), and (ii) an oblivious part that is executed under MPC (e.g., joins, aggregations). Some of these works report results for a

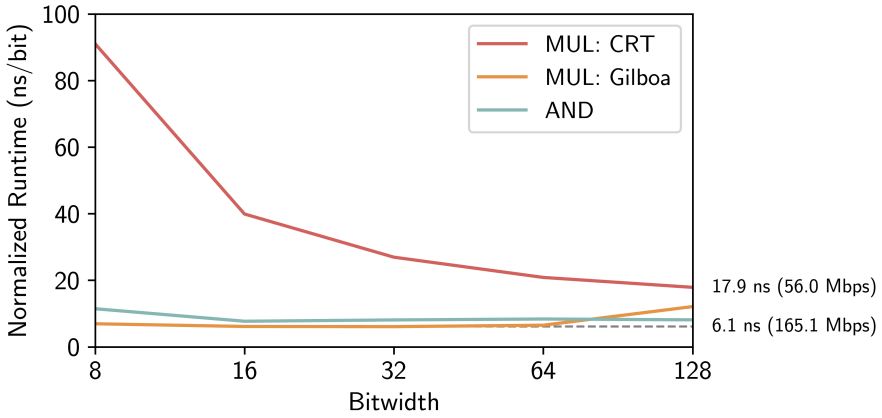

Figure 17: Normalized fastest runtime for our triple-generation implementations. Dashed line represents the minimum Gilboa time observed; it slows down slightly at 128 bits. Reciprocal of the values in Table 3.

small subset of TPC-H queries on a maximum input size of 10M rows per table [FCH+24] (including rows processed in plaintext). Senate [PKY+21] proposes a custom malicious-secure circuit decomposition protocol that supports any number of parties and provides strong security against dishonest majorities. Even though the codebase is not publicly available, published results indicate that the protocol's scalability is limited: the paper reports performance for a subset of TPC-H queries with 16k input rows (Scale Factor ∼0.004), and three queries from §8 (`Pwd`, `Comorb.`, and `Credit`) with up to 80k rows.

Contrary to Orq, join optimizations in the above works rely on the assumption that each party owns one or more input tables. For example, they can improve performance in a scenario where a hospital and an insurance company wish to identify common patients with high premiums, but they are not effective (or do not even apply) in cases where multiple hospitals contribute to the patients table. Orq's operators are designed for the more general outsourced setting, hence, they are independent of the number of participants and data schema. Moreover, while circuit decomposition techniques pay off in early stages of complex queries, later stages still need to be evaluated under MPC by all parties. As a result, Orq could also be used in the peer-to-peer setting to speed up the oblivious (and by far most expensive) part of complex queries. Other systems for outsourced MPC like Scape [HZF+22] (not publicly available) and Secrecy [LKFV23] report results for queries with up to two joins and maximum input sizes of 2M rows (for joins) and 8M rows (for other operators). Sharemind [BKK+16] (also not publicly available) reports results for queries with a single join and up to 17M input rows.

Oblivious join operators. Several works from the cryptography community study individual join operators in the outsourced setting, building from research into private set intersection (PSI). Mohassel et al. [MRR20] propose efficient one-to-one joins with linear complexity, but rely on the LowMC library that has been cryptanalyzed [LIM21]. For one-to-many joins, Krastnikov et al. [KKS20], Badrinarayanan et al. [BDG+22], and Asharov et al. [AHK+23] propose protocols based on sorting. They report results with up to 1M rows per input table and do not evaluate complex queries like in Orq. These operators have the same asymptotic costs as our join-aggregation operator, but are tailored to 3-party computation protocols [BDG+22, AHK+23] or target Trusted Execution Environments (TEEs) [KKS20]. DJoin [NH12] hides intermediate result sizes using differentially-private

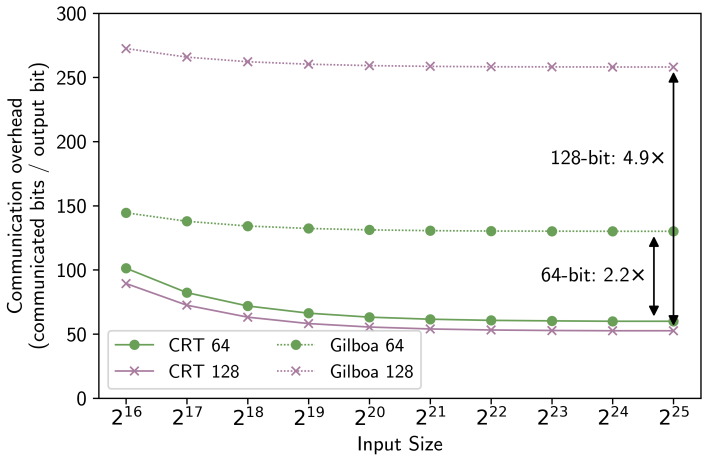

Figure 18: Normalized communication overhead of the two arithmetic triple algorithms, shown for 64 and 128 bit outputs.

noise, but is only able to evaluate cardinality queries with simple predicates. While the one-to-many join protocol of Badrinarayanan et al. [BDG+22] is reminiscent of ours (their "aggregation trees" are actually just prefix networks, like our AggNet), their many-to-many join protocol requires output-size leakage or a quadratic blowup. None of these works support many-to-many joins without suffering from the limitations we described in §1.

State-of-the-art systems for relational analytics in TEEs [ZDB+17, DLP+20, MWD+25] also provide oblivious join and group-by operators. To protect access patterns from side-channel attacks, join and aggregation in these systems rely on bitonic sort, which requires $O(n \log^2 n)$ operations. Early systems perform the grouping using fast parallel scans [ZDB+17], while recent approaches rely on prefix sums [MWD+25]. These TEE-based systems perform the join and the group-by one after the other. Consequently, they either leak intermediate result sizes [MWD+25], resort to worst-case padding [DLP+20], or require users to provide an upper bound on the query result size [ZDB+17]. In contrast, Orq uses a *composite* join-aggregation operator, with oblivious quicksort and an aggregation network, that requires $O(n \log n)$ operations in total. The techniques used by Orq in the MPC context can also be used to achieve oblivious execution in enclaves, and this is an exciting direction for future work.

10 Conclusion

We have presented Orq, a scalable, secure framework for evaluating complex relational analytics under MPC. We instantiate Orq in multiple threat models, contribute a unified join-aggregation operator to the literature, and implement state-of-the-art sorting protocols. We deploy Orq to obliviously execute queries with multi-way joins larger than any prior work. As presented, Orq requires data analysts to translate queries into our dataflow API; future work includes integrating Orq with an automatic query planner. Cryptographic advances in oblivious sorting will also directly lead to performance improvements in Orq.

Acknowledgements

We thank the anonymous reviewers and artifact evaluators for their constructive feedback that substantially improved the paper. We also thank Selene Wu for implementing low-level optimizations in the Orq runtime, and Sakshi Sharma for testing our artifact. This work was developed in part with computing resources provided by the Chameleon [KAZ+20] and CloudLab [DRM+19] research testbeds, supported by the National Science Foundation. Final results and evaluation used Amazon Web Services clusters through the CloudBank project (National Science Foundation Grant No. 1925001). This material is based upon work supported by the National Science Foundation under Grant No. 2209194, REU supplement awards No. 2432612 and 2326580, by the DARPA SIEVE program under Agreement No. HR-00112020021, and by a gift from Robert Bosch GmbH.

References

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 805–817, October 2016. URL: http://doi.acm.org/10.1145/2976749.2978331, doi:10.1145/2976749.2978331.

[AFO+21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 610–629, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484560.

[AHI+22] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 125–138, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560691.

[AHK+23] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. Secure statistical analysis on multiple datasets: Join and group-by. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3298–3312, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3623119.

[App21] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA). https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf, 2021. [Online; accessed April 2025].

[APRR24] Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. Improved alternating-moduli prfs and post-quantum signatures. In *Advances in Cryptology – CRYPTO 2024: 44th Annual International Cryptology Conference, CRYPTO 2024, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part VIII*, page 274–308, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-68397-8_9.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In *AsiaCCS*, pages 40–41. ACM, 2012.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, pages 307–314, 1968. URL: http://doi.acm.org/10.1145/1468075.1468121, doi:10.1145/1468075.1468121.

[Bau26] Eli Baum. To catch a cheater: Some notes on the malicious (in)security of ORQ. Cryptology ePrint Archive, Paper 2026/1152, 2026. URL: https://eprint.iacr.org/2026/1152.

[BBHL15] Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie Lumbroso. Mergeshuffle: a very fast, parallel random permutation algorithm. *arXiv preprint 1508.03167*, 2015.

[BBM+25] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. Orq: Complex analytics on private data with strong security guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 802–833, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3731569.3764833.

[BDG+22] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. Secret-shared joins with multiplicity from aggregation trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 209–222, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560670.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference (CRYPTO)*, pages 420–432, August 1991. doi:10.1007/3-540-46766-1_34.

[BEE+17] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017. URL: http://www.vldb.org/pvldb/vol10/p673-rogers.pdf, doi:10.14778/3055330.3055334.

[BHE+18] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320, 2018.

[BK82] RP Brent and HT Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, 1982.

[BKK+16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2016(3):117–135, 2016. URL: http://www.degruyter.com/view/j/popets.2016.2016.issue-3/popets-2015-0019/popets-2016-0019.xml.

[BKM20] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS*. The Internet Society, 2020.

[BLT14] Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems*, pages 59–74, Cham, 2014. Springer International Publishing.

[Bos24] Boston Women's Workforce Council. Data privacy: Ensuring secure and private data analysis. https://thebwwc.org/mpc, 2024.

[BPH+20] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment*, 13(12):2691–2705, 2020.

[BS26] Andreas Brüggemann and Thomas Schneider. When trying to catch cheaters breaks the mpc: Breaking and fixing delayed consistency checks in trident, fantastic four, swift, and quad. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–487. Springer, 2026.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–282, Boston, Massachusetts, USA, 2017. USENIX Association. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200. USENIX Association, 2021.

[DHIM25] Jack Doerner, Iftach Haitner, Yuval Ishai, and Nikolaos Makriyannis. From ot to ole with subquadratic communication. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 2249–2263, 2025.

[DLP+20] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *EuroSys*, pages 39:1–39:17. ACM, 2020.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[DRM+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[DRPS22] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2450–2468, 2022. doi:10.1109/SP46214.2022.9833611.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL:

https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation.

[EGK+20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO (2)*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852. Springer, 2020.

[FCH+24] Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, et al. Secretflow-scql: A secure collaborative query platform. *Proceedings of the VLDB Endowment*, 17(12):3987–4000, 2024.

[FY38] Ronald Aylmer Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, 1938.

[FZL+23] Muhammad Faisal, Jerry Zhang, John Liagouris, Vasiliki Kalavri, and Mayank Varia. TVA: A multi-party computation system for secure and expressive time series analytics. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5395–5412, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/faisal.

[GBLP96] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 152–159, 1996. doi:10.1109/ICDE.1996.492099.

[Gil99] Niv Gilboa. Two party rsa key generation. In *Annual International Cryptology Conference*, pages 116–129. Springer, 1999.

[GLMP18] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, pages 315–331. ACM, 2018.

[Goo11] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388. ACM, 2011.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.

[HSJ86] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HZF+22] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. Scape: Scalable collaborative analytics system on private database with malicious security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1740–1753, 2022. doi:10.1109/ICDE53745.2022.00176.

[JBA14] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2014.

[JKU11] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security (ACNS)*, June 2011.

[JPR16] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. AJAR: aggregations and joins over annotated relations. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 91–106. ACM, 2016. doi:10.1145/2902251.2902293.

[KAZ+20] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[KKM+22] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. Sok: Cryptanalysis of encrypted search with LEAKER - A framework for leakage attack evaluation on real-world data. In *EuroS&P*, pages 90–108. IEEE, 2022.

[KKM+24] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Jamie DeMaria, Andrew Park, and Amos Treiber. MAPLE: markov process leakage attacks on encrypted search. *Proc. Priv. Enhancing Technol.*, 2024(1):430–446, 2024.

[KKS20] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(11):2132–2145, 2020. URL: http://www.vldb.org/pvldb/vol13/p2132-krastnikov.pdf.

[KTKP24] Darya Kaviani, Sijun Tan, Pravein Govindan Kannan, and Raluca Ada Poda. Flock: A framework for deploying on-demand distributed trust. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[KVH+20] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[lC25] The libsodium Community. libsodium: A modern, portable, easy to use crypto library. https://libsodium.org/, 2025. [Online; accessed September 2025].

[LIM21] Fukang Liu, Takanori Isobe, and Willi Meier. Cryptanalysis of full lowmc and lowmc-m with algebraic techniques. In *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part III*, page 368–401, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-84252-9_13.

[Lin20] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, dec 2020. doi:10.1145/3387108.

[LKFV23] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1031–1056, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/liagouris.

[Lu05] Mi Lu. *Arithmetic and logic in computer systems*. John Wiley & Sons, 2005.

[LWDY24] Qiyao Luo, Yilei Wang, Wei Dong, and Ke Yi. Secure query processing with linear complexity. *arXiv preprint 2403.13492*, 2024. URL: https://arxiv.org/abs/2403.13492, arXiv:2403.13492.

[LWYY25] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. Bitgc: garbled circuits with 1 bit per gate. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 437–466. Springer, 2025.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. doi:10.1145/1061318.1061322.

[MH96] C.J.H. McDiarmid and R.B. Hayward. Large deviations for quicksort. *J. Algorithms*, 21(3):476–507, November 1996. doi:10.1006/jagm.1996.0055.

[Moz19] Mozilla. Next steps in privacy-preserving telemetry with Prio. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2019. [Online; accessed September 2025].

[MRR20] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and psi for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1271–1287, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3423358.

[MWD+25] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. OBLIVIATOR: Oblivious parallel joins and other operators in shared memory environments. Cryptology ePrint Archive, Paper 2025/183, 2025. URL: https://eprint.iacr.org/2025/183.

[MZF+23] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. SecretFlow-SPU: A performant and User-Friendly framework for Privacy-Preserving machine learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 17–33, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/atc23/presentation/ma.

[NH12] Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 149–162, October 2012. URL: http://dl.acm.org/citation.cfm?id=2387880.2387895.

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, 2015. doi:10.1145/2810103.2813651.

[NL24] Thomas Neumann and Viktor Leis. A critique of modern SQL and a proposal towards a simple and expressive query language. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL: https://www.cidrdb.org/cidr2024/papers/p48-neumann.pdf.

[ORSS08] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 267–273, USA, 2008. USENIX Association.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2(02n03):205–211, 1992.

[PHP+24] Xinyu Peng, Feng Han, Li Peng, Weiran Liu, Zheng Yan, Kai Kang, Xinyuan Zhang, Guoxing Wei, Jianling Sun, and Jinfei Liu. Mapcomp: A secure view-based collaborative analytics framework for join-group-aggregation. *arXiv preprint 2408.01246*, 2024. URL: https://arxiv.org/abs/2408.01246, arXiv:2408.01246.

[PKY+21] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/poddar.

[PRRS24] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Efficient permutation correlations and batched random access for two-party computation. Cryptology ePrint Archive, Paper 2024/547, 2024. URL: https://eprint.iacr.org/2024/547.

[RR24] Peter Rindal and Lance Roy. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe, Last access: September 2024.

[RZCGP24] Mayank Rathee, Yuwen Zhang, Henry Corrigan-Gibbs, and Raluca Ada Popa. Private analytics via streaming, sketching, and silently verifiable proofs. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 194–194. IEEE Computer Society, 2024.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

[SQL25] SQLite. Sqlite sql database engine. https://sqlite.org/, 2025. [Online; accessed September 2025].

[Tra24] Transaction Processing Performance Council. TPC-H Benchmark Specification. https://www.tpc.org/tpch/, 2024. [Online; accessed September 2024].

[Uni23] United Nations Global Working Group (GWG) Task Team on Privacy Preserving Techniques. Case study repository. https://unstats.un.org/wiki/display/UGTTOPPT/Case+study+repository, 2023.

[VSG+19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In George Candea, Robbert van Renesse, and Christof Fetzer, editors,

Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019, pages 3:1–3:18. ACM, 2019. doi:10.1145/3302424.3303982.

[WY21] Yilei Wang and Ke Yi. *Secure Yannakakis: Join-Aggregate Queries over Private Data*, page 1969–1981. Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3448016.3452808.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[YGI09] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 247–260, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1629575.1629600.

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–298, Boston, Massachusetts, USA, 2017. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

[ZGY+24] Wenhao Zhang, Xiaojie Guo, Kang Yang, Ruiyu Zhu, Yu Yu, and Xiao Wang. Efficient actively secure DPF and ram-based 2pc with one-bit leakage. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 561–577. IEEE, 2024. doi:10.1109/SP54263.2024.00205.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. doi:10.1145/2934664.

Appendix

In the following sections, we provide additional details on our oblivious shuffling (Appendix A), radixsort (Appendix B), quicksort (Appendix C), and join primitives (including arguments of correctness and more information on the trimming heuristic; Appendix D). Appendix E provides detailed bandwidth measurements for Orq and the systems we compare against. Appendices are not included in the peer-review process.

A Oblivious Shuffle

This section describes a set of primitives related to obliviously shuffling a secret-shared vector. Existing works typically discuss such primitives for a fixed number of parties and specific MPC protocol (e.g., [AHI+22, PRRS24]). Our primary contribution in this section is a single stack of primitives that can be used across a variety of MPC protocols and threat models. We also provide novel algorithms for obliviously inverting elementwise permutations and converting elementwise permutations between arithmetic and boolean sharings.

A.1 Preliminaries

A permutation is a bijective mapping from $[n]$ to $[n]$, where $[n]$ represents the set $\{1, \ldots, n\}$. We denote the composition of permutations π and ρ by $\pi \circ \rho(\cdot) = \pi(\rho(\cdot))$. We represent permutations as index maps: starting from a data array $\vec{x}$, if $\pi_i = j$ then $\pi(\vec{x})$ maps the value x_i to position j. As a special case, a *sorting permutation* for an input vector $\vec{x}$ is a permutation σ such that $\sigma(\vec{x})$ is sorted. We use $\mathbb{I}$ to denote the identity permutation $\mathbb{I} = (1, \ldots, n)$. We use $[\![x]\!]$ to denote that a value x is secret-shared, and we do not distinguish in notation between an arithmetic and boolean sharing.

Like Asharov et al. [AHI+22], in this work we consider two different forms of secret-sharing for permutations:

Elementwise Permutation. A vector of secret-shared indices, denoted $[\![\pi]\!] = ([\![\pi_1]\!], \ldots, [\![\pi_n]\!])$. The elements can either be arithmetic or boolean secret-sharings.

Sharded Permutation. A composition of random permutations, denoted $\langle \pi \rangle = \pi_n \circ \cdots \circ \pi_1$.

We reproduce an important fact about permutations from Observation 2.4 of Asharov et al. [AHI+22].

Fact 1. *Let π and σ be permutations over the set $[m]$ and let $\vec{\sigma} = (\sigma(1), \ldots, \sigma(m))$ be the vector of destinations of the permutation σ. Then $\pi(\vec{\sigma}) = \sigma \circ \pi^{-1}([m])$, i.e. a vector of destinations for the permutation $\sigma \circ \pi^{-1}$.*

A.2 Local permutations

Many of the oblivious shuffling primitives that will be discussed in the following sections involve locally generating and locally applying random permutations as subroutines. In this section, we discuss the algorithms used for generating and applying random local permutations and our methods for parallelizing these operations.

Generating random local permutations. To generate local random permutations, we use the Fisher-Yates shuffle algorithm [FY38], a single-threaded algorithm. While there exist algorithms for generating random permutations in parallel (such as MergeShuffle [BBHL15]), we typically need to generate many random permutations at once. As a result, we parallelize the generation by distributing a batch of permutations among all cores, so that each permutation is generated in a single-threaded fashion.

We may additionally wish to generate identical random permutations among multiple parties. This can be achieved by having each party generate the permutation locally using a pseudorandom generator (PRG) with the same seed.

Applying random local permutations. Unlike generation, we apply random local permutations one at a time, so we need to parallelize each individual local permutation application. To apply a local permutation, we want each element x_i of a vector $\vec{x}$ to be placed at index π_i in the output vector $\vec{y}$. We can easily divide $\vec{x}$ and π into contiguous blocks and give one to each thread. Since we must access random elements in $\vec{y}$, we cannot give each thread a contiguous block of $\vec{y}$. However, we can give each thread full write access to $\vec{y}$ and mathematically guarantee that no element will be written to more than once.

A.3 Sharded permutation protocols

In this section, we describe protocols to generate and apply sharded permutations, which we denote as `genSharded- Perm` and `applyShardedPerm`, respectively. These permutations apply to all three MPC protocols used in Orq. We explain separately our protocols in the honest-majority and dishonest-majority settings.

Honest majority. In the three-party setting with replicated secret-sharing, we use the methods for generating and applying sharded permutations by Asharov et al. [AHI+22], which we restate below for completeness. A sharded permutation $\langle\pi\rangle = \pi_3 \circ \pi_2 \circ \pi_1$ can be expressed as a replicated secret sharing

$$\langle\pi\rangle = ((\pi_1, \pi_2), (\pi_2, \pi_3), (\pi_3, \pi_1)).$$

That is, each party P_i holds two permutations: one in common with P_{i+1} and one in common with P_{i-1}. To generate such a sharing, each pair of parties generates a random local permutation using their common PRG seed. However, there is one permutation that P_i does not know, so π remains secret.

Later when the parties call `applyShardedPerm` to apply a sharded permutation, each pair of parties locally applies their common permutation and reshares the permuted result to the excluded party. First, P_1 and P_3 permute under π_1 and reshare to P_2, and then so on for π_2 and π_3.

We generalize the protocol to work with any replicated secret-sharing scheme in the honest-majority setting, with an emphasis on the four-party Fantastic Four [DEK21] protocol. The three-party protocol proceeds in a sequence of rounds, where in each round a subset of the parties — which we will call a *shuffle group* — locally permutes the vector and reshares it to the remaining party. We generalize this idea of shuffle groups and design protocols in which each round contains a local permutation application and resharing by a single shuffle group. For security, we require that shuffle groups obey two properties:

1. Each shuffle group can collectively hold a sharing of the input data. As a result, the size of a shuffle group is greater than T, which is why this approach is limited to the honest-majority setting.

2. There must exist at least one shuffle group containing no corrupted parties, so that the composed permutation π is unknown to the adversary.

Using the Fantastic Four protocol as a concrete example [DEK21], here is a set of shuffle groups in the semi-honest setting: $\mathcal{G} = \{\{P_0, P_1\}, \{P_2, P_3\}\}$. Since there is only $T = 1$ corrupted party, each shuffle group possesses at least one copy of all secret shares, and there is a shuffle group with no corrupted parties.

There are two ways to extend this idea to the malicious-secure four-party setting. One trivial approach is to combine the above idea with the generic compiler from semi-honest to malicious security for the `reshare` functionality proposed by Asharov et al. [AHI+22]. However, we take a different approach that provides ease of implementation and consistency with the other functionalities in the Fantastic Four protocol [DEK21]. We use four shuffle groups of three parties each, so that each share is contained twice in each group, allowing for redundancy in resharing the value to the excluded party:

$$\mathcal{G} = \{\{P_0, P_1, P_2\}, \{P_1, P_2, P_3\}, \{P_2, P_3, P_0\}, \{P_3, P_0, P_1\}\}.$$

We can then base the malicious security of the oblivious sharded permutation application on the black-box security of the `INP` protocol described in Fantastic Four [DEK21]. Specifically, the receiving party receives either two copies of each share or one copy and its hash. In either case, a single corrupted party cannot corrupt both values received, so cheating will be detected with overwhelming probability.

Dishonest majority. In the two-party setting with one dishonest party, we cannot use shuffle groups and therefore instead adopt the framework of Peceny et al. [PRRS24] for generating and applying sharded permutations. Their approach is based on a *permutation correlation*, which is a pair of tuples, one held by each party: $(A, B_0), (B_1, \pi)$ such that $\pi(A) = B_0 + B_1$. Here, $A, B_0, B_1 \in \mathbb{F}^n$ and π is a random permutation over n elements. B_0 and B_1 can be either arithmetic or boolean secret-shares, with the addition operator defined correspondingly.

To generate a sharded permutation, it suffices to generate two permutation correlations in a preprocessing phase, one with each party as the sender. Peceny et al. [PRRS24] propose two protocols for generating permutation correlations, both based on oblivious pseudorandom functions (OPRFs), with one of them additionally using pseudorandom correlation generators to achieve communication sublinear in the length of the elements (as opposed to the length of the permutations).

To apply a sharded permutation obliviously, we use the protocol $\Pi_{\text{Comp}-\text{Perm}}$ from Peceny et al. [PRRS24]. We make use of their library's OPRF implementation [APRR24, PRRS24] and construct the rest of the permutation correlations from scratch. Similar ideas can be used to build `applyInverseShardedPerm` that applies the inverse of a sharded permutation. Since the output of the OPRF is a boolean secret sharing, we generate all permutation correlations with a boolean sharing and convert to an arithmetic sharing where needed using a boolean-to-arithmetic conversion protocol.

A.4 Shuffling framework

Our framework provides generic interfaces to the functionalities `genShardedPerm`, `applyShardedPerm`, and `applyInverseShardedPerm` that call the protocols to generate and apply (respectively) a sharded permutation as described in the previous section depending on the desired MPC protocol and threat model. In this section, we describe our abstract interface and the protocols for oblivious shuffling primitives. A detailed analysis of the complexity of each protocol in each setting can be found in Table 4.

The Permutation Manager abstraction. We now describe the `PermutationManager`, our abstraction for generating sharded permutations in a setting-agnostic manner. The goal is to make all differences in the generation and application of sharded permutations invisible to the higher level shuffle protocols. The `PermutationManager` exposes two functions:

- `genShardedPerm<T>(size, enc)`
- `genShardedPermPair<T1,T2>(size, enc1, enc2)`

The function `genShardedPerm` returns a random sharded permutation over `size` elements of type `T` (e.g., `int32`), where the elements have encoding `enc` (either an arithmetic or boolean sharing). The function `genShardedPermPair` returns two random sharded permutations over `size` elements such that they correspond to the same random permutation. This is necessary if we wish to permute multiple columns of data according to the same sharded permutation, as is the case, for example, when applying or composing elementwise permutations. The two vectors have types `T1` and `T2` and encodings `enc1` and `enc2`. Both functions allow for batched, parallel generation of many sharded permutations. Since all computation is data-independent, our framework generates sufficiently-many sharded permutations in preprocessing; when called, these functions fetch an already-generated permutation.

In the honest-majority setting, sharded permutations are identical for all input types, regardless of the bitwidth or sharing type of the vectors to be permuted. To reserve a

pair of sharded permutations which represent the same permutation, we simply generate a single sharded permutation $\langle\pi\rangle$ and return the pair $(\langle\pi\rangle, \langle\pi\rangle)$. The need to distinguish between a single sharded permutation and a pair of sharded permutations only arises in the dishonest-majority setting, as the underlying permutation correlations are type-dependent and cannot be securely reused.

Applying and composing permutations. As described in Section A.3, our framework provides methods that apply a sharded permutation and its inverse. It also provides methods for oblivious shuffling, permutation composition, and applying an elementwise permutation.

Our oblivious shuffle protocol `shuffle` simply generates and applies a sharded permutation, as shown in Protocol 8.

Protocol 8: Π_{Shuffle} : `shuffle`

input : $[\![x]\!]$
output : $[\![\pi(x)]\!]$ for random π

1 $\langle\pi\rangle \leftarrow$ `genShardedPerm<type`(x)`>(size`(x), `enc`$(x))$
2 $[\![\pi(x)]\!] \leftarrow$ `applyShardedPerm`$([\![x]\!], \langle\pi\rangle)$
3 **return** $[\![\pi(x)]\!]$

Next, Protocol 9 shows `applyElementwisePerm`, which generalizes protocol 4.2 of Asharov et al. [AHI+22]. The main idea here (which we use also in subsequent protocols) is that if we obliviously shuffle the elementwise permutation, then it is safe to open the shuffled vector in line 6 while revealing nothing about the unshuffled vector.

Protocol 9: $\Pi_{\text{ApplyElem}}$: `applyElementwisePerm`

input : $[\![x]\!], [\![\rho]\!]$
output : $[\![\rho(x)]\!]$

1 $\tau_1, \tau_2 \leftarrow$ `type`(x), `type`(ρ)
2 $\epsilon_1, \epsilon_2 \leftarrow$ `enc`(x), `enc`(ρ)
3 $\langle\pi_1\rangle, \langle\pi_2\rangle \leftarrow$ `genShardedPermPair<`τ_1, τ_2`>(size`$(x), \epsilon_1, \epsilon_2)$
4 $[\![\pi(x)]\!] \leftarrow$ `applyShardedPerm`$([\![x]\!], \langle\pi_1\rangle)$
5 $[\![\pi(\rho)]\!] \leftarrow$ `applyShardedPerm`$([\![\rho]\!], \langle\pi_2\rangle)$
6 $\pi(\rho) \leftarrow$ `open`$([\![\pi(\rho)]\!])$
7 $[\![\rho(x)]\!] \leftarrow$ `localApplyPerm`$([\![\pi(x)]\!], \pi(\rho))$
8 **return** $[\![\rho(x)]\!]$

Finally, Protocol 10 shows `composePerms`, our method to compose two permutations $[\![\sigma]\!]$ and $[\![\rho]\!]$ that is based on protocol 4.3 of Asharov et al. [AHI+22]. For simplicity, we describe the case where both permutations have the same encoding (either arithmetic or boolean). Otherwise, we can convert one encoding using the conversion protocol described next. Alternatively, we could allow the input elementwise permutations to have different sharing types, but we would then have to pay the additional cost of generating a pair of sharded permutations rather than a single sharded permutation.

Novel shuffle primitives. We propose two novel protocols related to oblivious shuffling. The first converts an elementwise permutation from either an arithmetic sharing to a boolean sharing or from a boolean sharing to an arithmetic sharing. The second inverts an elementwise permutation.

To convert between arithmetic and boolean encodings, one trivial approach is to use arithmetic to boolean share conversions for each element of the permutation. In the

Protocol 10: Π_{Compose} : `composePerms`

input : $[\![\sigma]\!], [\![\rho]\!]$
output : $[\![\rho \circ \sigma]\!]$

1 $\langle \pi \rangle \leftarrow$ `genShardedPerm<type(`σ`)>(size(`σ`), enc(`σ`))`
2 $[\![\pi(\sigma)]\!] \leftarrow$ `applyShardedPerm(`$[\![\sigma]\!], \langle \pi \rangle$`)`
3 $\pi(\sigma) \leftarrow$ `open(`$[\![\pi(\sigma)]\!]$`)`
4 $[\![\pi \circ \sigma^{-1}(\rho)]\!] \leftarrow$ `localApplyPerm(`$[\![\rho]\!], (\pi(\sigma))^{-1}$`)`
5 $[\![\rho \circ \sigma]\!] \leftarrow$ `applyInverseShardedPerm(`$[\![\pi \circ \sigma^{-1}(\rho)]\!], \langle \pi \rangle$`)`
6 **return** $[\![\rho \circ \sigma]\!]$

Table 4: Communication and round complexities for 2PC, 3PC, and 4PC shuffling primitives. Here, n denotes the number of elements in the input vector, ℓ denotes the bitwidth of the elements, and ℓ_σ denotes the bitwidth of a permutation ($\ell_\sigma = 32$ in our system).

	2PC		3PC		4PC	
Primitive	**Comm.**	**Rounds**	**Comm.**	**Rounds**	**Comm.**	**Rounds**
`genSharded`	preproc.	preproc.	-	-	-	-
`applySharded`	$2\ell n$	2	$6\ell n$	3	$24\ell n$	4
`shuffle`	$2\ell n$	2	$6\ell n$	3	$24\ell n$	4
`applyElementwise`	$2\ell n + 3\ell_\sigma n$	5	$6\ell n + 7\ell_\sigma n$	7	$24\ell n + 25\ell_\sigma n$	9
`compose`	$5\ell_\sigma n$	5	$13\ell_\sigma n$	7	$49\ell_\sigma n$	9
`invertElementwise`	$5\ell_\sigma n$	5	$13\ell_\sigma n$	7	$49\ell_\sigma n$	9
`convertElementwise`	$5\ell_\sigma n$	5	$13\ell_\sigma n$	7	$49\ell_\sigma n$	9

dishonest-majority setting, we use this trivial approach. However, in the honest-majority setting, we can take advantage of the additional structure imposed by permutations by recognizing that we already know every value contained in the plaintext vector, but not the ordering. Specifically, we can open a shuffled version of the vector, re-share the opened vector under the desired type, and obliviously apply the inverse of the shuffle. This approach, shown in Protocol 11, is faster than the trivial protocol in the honest-majority.

Protocol 11: Π_{Conv} : `convertElementwisePerm`

input : $[\![x]\!]_T$, $T \in \{A : 0, B : 1\}$
output : $[\![x]\!]_{1-T}$

1 $\tau \leftarrow$ `type(`x`)`
2 $\langle \pi_1 \rangle, \langle \pi_2 \rangle \leftarrow$ `genShardedPerm<`τ, τ`>(`$x, T, 1-T$`)`
3 $[\![\pi(x)]\!]_T \leftarrow$ `applyShardedPerm(`$[\![x]\!]_T, \langle \pi_1 \rangle$`)`
4 $\pi(x) \leftarrow$ `reveal(`$[\![\pi(x)]\!]_T$`)`
5 $[\![\pi(x)]\!]_{1-T} \leftarrow$ `secretShare(`$\pi(x), 1-T$`)`
6 $[\![x]\!]_{1-T} \leftarrow$ `applyInverseShardedPerm(`$[\![\pi(x)]\!]_{1-T}, \langle \pi_2 \rangle$`)`
7 **return** $[\![x]\!]_{1-T}$

We now describe a protocol for inverting an elementwise permutation. Whereas in `applyInverseShardedPerm` the inverse was not computed directly but applied to a vector, here we wish to compute the inverse so it can be composed with other elementwise permutations. The protocol Π_{Inv} is simple: to invert an elementwise permutation, we simply obliviously apply it to the identity permutation. Let $[\![\pi]\!]$ be the permutation

we wish to invert. We create an identity vector $\mathbb{I} = (1, \ldots, n)$ and secret-share it. We then obliviously apply $[\![\pi]\!]$ to $[\![\mathbb{I}]\!]$. The permuted identity vector is π^{-1}. By Fact 1, $\pi(\mathbb{I}) = \mathbb{I} \circ \pi^{-1} = \pi^{-1}$. We provide a formal description in Protocol 12, and we discuss how to use such an inversion protocol to extract the sorting permutation from a sorting protocol in section 5.3.

Protocol 12: Π_{Inv} : `invertElementwisePerm`

input : $[\![\pi]\!]$
output : $[\![\pi^{-1}]\!]$
1 $[\![\pi^{-1}]\!] \leftarrow$ `applyElementwisePerm`$([\![\mathbb{I}]\!], [\![\pi]\!])$
2 **return** $[\![\pi^{-1}]\!]$

B Radixsort analysis

Our radixsort is similar to the protocol of Bogdanov et al. [BLT14], except we make use of recent advances in oblivious shuffling discussed in §A.4. For reasonable bitwidths like $\ell = 32$, our protocol can be seen as an optimization of Asharov et al. [AHI+22] that achieves a significant improvement in round complexity and a mild improvement in communication complexity.

Comparison. The key difference between our works is that, whereas Asharov et al. runs `composePerms` after sorting each bit, we apply the permutation for each bit to a larger vector. While this increases the communication required for the permutation application step, it does not add additional rounds and it allows us to eliminate the compose step and therefore reduce the round complexity. We demonstrate analytically in Table 5 and empirically in Figure 19 that our protocol typically outperforms Asharov et al. [AHI+22] for both $\ell = 32$ and $\ell = 64$ bits. Because the codebase of Asharov et al. [AHI+22] is proprietary, we provide our own reimplementation of their protocol that we use in the Figure 19 benchmarks.

This may seem contradictory to the claims of Asharov et al. In fact, they claim that it is precisely the permutation composition step that leads to their improvement in performance over Bogdanov et al. The distinction is that the protocol of Asharov et al. has constant factors that scale better than Bogdanov et al. (and better than our protocol) as ℓ grows without bound, whereas our analysis focuses on the values $\ell = 32$ and 64 that we use in our shuffling and sorting frameworks.

Relationship to Bogdanov et al. [BLT14]. Here we show that by instantiating the components of the Bogdanov et al. protocol with the more efficient variants proposed in recent works, we arrive at our protocol. Concretely, there are three differences in our protocols. First, their work converts the bit to sort into an arithmetic sharing and subsequently computes the $[\![ord]\!]$ vector, whereas our work and Asharov et al. do the same using the `genBitPerm` subprotocol. Second, they apply the $[\![ord]\!]$ permutation to the input vector using an oblivious shuffle, open, and local application, which we do with our equivalent `applyElementwisePerm` protocol (Protocol 4.2 of Asharov et al. in the three-party setting). Finally, the input padding and permutation extraction procedure described in Section 5 of Bogdanov et al. is equivalent to our procedure in §5.3 for padding and permutation extraction, plus our protocol is generalized to include descending-order sorting.

Performance analysis. In the remainder of this section and in Table 5, we compare the costs of our protocol and Asharov et al. [AHI+22], specifically in the three-party setting as that is the only one they support.

Common elements. Both protocols invoke `genBitPerm` ℓ times. Each invocation incurs n calls to `b2abit` and n multiplications that cost $3\ell_\sigma n$ and $\ell_\sigma n$ bits of communication, and 3 and 1 rounds of communication, respectively, with elements of size ℓ_σ since `genBitPerm` computes a permutation. In total, each protocol communicates $4\ell_\sigma n$ bits over 4 rounds for each call to `genBitPerm`, amounting to $4\ell \cdot \ell_\sigma n$ bits of communication and 4ℓ rounds in total.

Costs of Asharov et al. There are two additional costs in Asharov et al. [AHI+22]: $\ell - 1$ calls to `applyElementwisePerm` and $\ell - 1$ calls to `composePerms`. First, each call to `applyElementwisePerm` operates over a single bit; using Table 4, we see that the total cost is $(\ell - 1)(6n + 7\ell_\sigma n)$ bits of communication over $7(\ell - 1)$ rounds. Second, the calls to `composePerms` collectively cost $13(\ell - 1)\ell_\sigma n$ bits of communication and $7(\ell - 1)$ rounds of communication. When adding the cost of the ℓ calls to `genBitPerm`, the total cost of the protocol is

$$24\ell \cdot \ell_\sigma n - 20\ell_\sigma n + 6(\ell - 1)n$$

bits of communication over $18\ell - 14$ rounds.

Our costs. In our protocol, we eliminate the permutation compositions and instead apply the permutations to a vector of bitwidth $\ell + \ell_\sigma$. We make:

- $\ell - 1$ calls to `applyElementwisePerm`, each costing $6(\ell + \ell_\sigma)n + 7\ell_\sigma n$ bits of communication and 7 rounds.
- One call to each of `convertElementwisePerm` and `invertElementwisePerm`, adding $26\ell_\sigma n$ bits of communication and 14 rounds of communication.

Therefore, in total, our protocol has a cost of

$$17\ell \cdot \ell_\sigma n + 13\ell_\sigma n + 6\ell^2 n - 6\ell n$$

bits of communication and $11\ell + 7$ rounds of communication.

Table 5: Cost analysis of the state-of-the-art radixsort protocol by Asharov et al. [AHI+22] and our hybrid radixsort protocol.

	Asharov et al. [AHI+22]	**Ours**
Asymptotic Comm.	$O(\ell n \log n)$	$O(\ell^2 n)$
Concrete Comm.	$[\ell_\sigma(24\ell - 20) + 6(\ell - 1)]n$	$[\ell_\sigma(17\ell + 13) + 6\ell(\ell - 1)]n$
Asymptotic Rounds	$O(\ell)$	$O(\ell)$
Concrete Rounds	$18\ell - 14$	$11\ell + 7$
$\ell = 1$ *Comm.*	$128n$ (4 rounds)	$960n$ (18 rounds)
$\ell = 32$ *Comm.*	$24122n$ (562 rounds)	$23776n$ (359 rounds)
$\ell = 64$ *Comm.*	$48890n$ (1138 rounds)	$59424n$ (711 rounds)

Discussion. In Table 5, we describe the costs both as a function of ℓ and ℓ_σ and for three values of the bitwidth. Our protocol has the better concrete round complexity, scaling as 11ℓ rather than 18ℓ (although they are asymptotically equivalent). For communication complexity, because ℓ and $\log n$ are often similar, the two protocols are typically asymptotically equivalent. Concretely, our protocol has the smaller constant factor for the $\ell \cdot \ell_\sigma$ term, which for typical values of ℓ close to ℓ_σ (e.g., $\ell = \ell_\sigma = 32$) means our protocol will

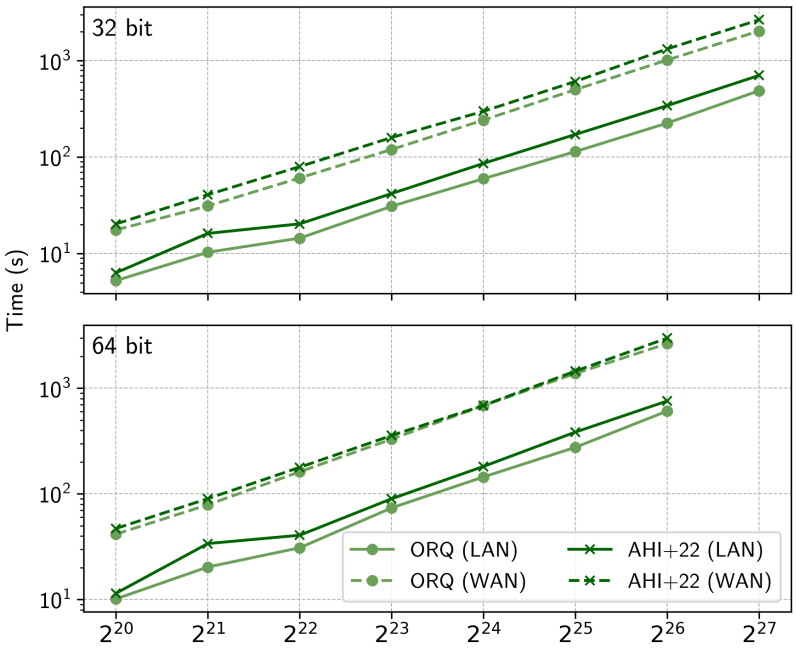

Figure 19: Comparison of our radixsort protocol with Asharov et al. [AHI+22] for (a) $\ell = 32$ and (b) $\ell = 64$. All data points are the average of 3 runs and are run with 34 threads in the 3-party setting. WAN data points are collected in a WAN environment with 20ms latency. 64-bit radixsort runs out of memory for 2^{27} input rows. Our hybrid protocol wins in all scenarios by up to 1.44×.

require less communication. However, only our protocol has an ℓ^2 term, so it is a poor choice when $\ell \gg \ell_\sigma$.

We discuss a few specific examples that are shown in Table 5 for the setting $\ell_\sigma = 32$ (since we never sort more than 4 billion elements due to speed constraints).

- If $\ell = 32$ as well, then our protocol saves a modest 1.4% improvement in total communication and a significant 36% improvement in communication rounds. In practice, the improved round complexity leads to substantially better performance, which we demonstrate empirically in both the LAN and WAN settings in Figure 19.
- If $\ell = 64$, then our protocol requires 22% more communication but decreases the number of rounds by 37%. Once again, Figure 19 shows empirically that Orq performs better in both the LAN and WAN settings when $\ell = 64$, although by a slimmer margin than with $\ell = 32$.
- If $\ell = 1$ then the constant additive overhead of input padding and permutation extraction makes our protocol inferior to Asharov et al.

C Bounding preprocessing for Quicksort

In the two-party SH-DM setting, we need to preprocess Beaver triples for the quicksort comparisons. Quicksort involves a nondeterministic number of comparisons, but we would like to probabilistically bound the running time so we can generate sufficient triples ahead of tim (generating triples in a large batch is substantially faster than generate multiple

smaller batches). For n input elements, we generate Beaver triples for $2n \lg n$ comparisons, which is sufficient approximately 99.9% of the time. We make use of the work of McDiarmid et al. [MH96], which discusses this problem in the case of a random pivot selection. Since we shuffle the input list, our method of pivot selection chooses a random element, so the analysis applies.

The expected number of comparisons q_n for quicksort with uniformly random pivot selection as $n \to \infty$ is

$$\begin{aligned} q_n &= 2n \ln n - (4 - 2\gamma)n + 2 \ln n + O(1) \\ &\leq 2n \ln n = (\ln 4)\, n \lg n \leq 1.39 n \lg n. \end{aligned}$$

This says nothing about how concentrated the distribution is around this expectation. Let Q_n be the random variable representing the actual number of comparisons required for a given run of quicksort. McDiarmid et al. [MH96] discuss the probability $p = \Pr[|Q_n/q_n - 1| > \varepsilon]$ describing the concentration. Here, $(\varepsilon + 1)$ is the *multiplicative* overhead over the expectation q_n. We want to solve for ε as a function of both the input size n and the probability p. We set the probability to a constant $p = 2^{-10}$. This is significantly larger than a typical statistical failure, because if we "fail," we can simply generate more triples in the online phase with no impact on security and an acceptable performance penalty. The chosen value of p makes our generated randomness sufficient for 99.9% of executions.

Theorem 1 of McDiarmid et al. [MH96] gives the following expression of the above probability.

$$p = n^{-2\varepsilon(\ln(\ln(n)) - \ln(\frac{1}{\varepsilon}) + O(\ln^{(3)}(n)))} \leq n^{-2\varepsilon(\ln(\ln(n)) - \ln(\frac{1}{\varepsilon}))}$$

To write ε as a function of p, we can take the natural log of both sides of this inequality. Even so, it is not feasible to calculate an analytic solution to this equation due to the presence of both ε and $\ln(1/\varepsilon)$ terms. To simplify the equation, we use the bound $\varepsilon \geq 0.43$ because $1.39 \cdot 1.43 \approx 2$. We simplify the equation using this value.

$$\varepsilon \leq -\frac{\ln(p)}{2\ln(n)(\ln(\ln(n)) - \ln(1/0.43))}$$

For $p = 2^{-10}$, $\varepsilon \leq 0.43$ for all $n \geq 1300$, meaning the chosen value of $\varepsilon = 0.43$ is suitable for all $n \geq 1300$. To offset the fact that values of $n < 1300$ are not covered by this case (however unlikely it may be for us to need a sort that small in practice), we handle this case separately with an additive buffer of $10,000$ triples whenever the input size is below $n = 2000$. This buffer is loose but sufficient, and it will not have a noticeable impact on performance for any reasonable use of the library.

D Analysis of the Join Operator

In this appendix, we address the security of our join operator and then describe its correctness in detail.

Security. Security comes naturally from our oblivious building blocks: since we never open any secret-shared data, and make no assumptions about the true cardinality of input or output tables, any execution of the join operator on tables L_1, R_1 is identically distributed to any other execution on similarly-sized tables L_2, R_2; $|L_1| = |L_2|$ and $|R_1| = |R_2|$. Note that here we refer to the observable size of the table in memory (both rows and columns) and not the number of valid rows (which would be the "size" of the table in the conventional sense). For example, even if L_2 and R_2 consisted entirely of (secret-shared) zero values, an execution of the join protocol would be indistinguishable from an execution on real values.

In more detail, we can analyze the security of the join operator using the arithmetic black-box model (e.g., [EGK+20, §2.1]), which provides an ideal functionality abstraction of all primitive MPC protocols for secret-sharing and reconstructing secrets, performing arithmetic and boolean operations, and converting between arithmetic and boolean shares. Through inspection of Protocols 2, 6, and 7, we can see that the join operators in Orq make use of MPC primitives in a black-box way. Concretely, the join-aggregation in Protocol 7 only uses oblivious logical operators (e.g., `AND` and `NOT`), (in)equality comparisons to find distinct keys, and calls to underlying methods like AggNet, TableSort, and Concat. These methods, in turn, are shown in Protocols 2–7 to rely exclusively on oblivious comparisons and logical operations along with the oblivious shuffling and sorting protocols defined in Appendix A and §5. Finally, inspection of all shuffling and sorting protocols (in Protocols 8–5) reveals that they only rely on oblivious arithmetic and boolean operators. Importantly, none of these algorithms opens sensitive data; the only openings are on permuted data (Protocols 9–10) or control flow data (Protocol 3) that have been carefully constructed so that the distribution of openings is independent of the data. As a result, Orq join operators inherit the security guarantees of the underlying MPC protocols.

Correctness. In the remainder of this section, we argue the correctness of the inner join operation and its variants. We will use the following important fact when reasoning about joins:

Fact 2. *Invalid rows are never revalidated.*

This fact is guaranteed by only operating on the valid column using logical AND. Thus, rows may move from valid to invalid ($\texttt{Valid} \wedge 0 \rightarrow \texttt{Invalid}$) but once invalidated can never be revalidated ($\forall x : \texttt{Invalid} \wedge x \rightarrow \texttt{Invalid}$). Rows are only validated when creating a *fresh* table.

Theorem 1 (Informal)**.** *The inner join protocol is correct.*

Proof Sketch. Define the key and valid columns of each input table as $L.k, L.V; R.k, R.V$, respectively. Without loss of generality we only consider a single abstract data column, $L.a$ and $R.b$, in each of the two tables. Assume the left table L has unique keys — each distinct value of k occurs at most once — and R has an arbitrary key distribution. Note that primary-key foreign-key relations are a strict subset of unique-key many-key relations, since keys on the right in a one-to-many relation need not exist on the left. Finally, while we discuss the case of a single key k below, our protocol transparently handles compound keys, $k_1 \mid\mid k_2 \mid\mid \ldots$, exactly as if all such key columns were concatenated into one.

The first step of the protocol is to concatenate the two tables. We concatenate by merging the schemas of the two tables and copying each table's rows into the appropriate columns. All other values are zero (such as within left-table rows but under right-table columns). We additionally append a T_{id} column, which is 0 for left-table rows and 1 for right-table rows, and sort by $V \mid\mid k \mid\mid T_{id}$. Any invalid rows in either table would remain invalid after concatenation and thus be sorted to the top of the table. The resulting table $O = \textsc{Concat}(L, R)$ may have the form:

Next we discuss the two main functions of our inner join algorithm: (a) identifying matching rows, according to the join keys, and (b) copying data into the result of the join.

We first turn to the identification of matching rows. This step takes the form of invalidating rows which do not belong to the output of the join. In the case of an inner join, we only keep those (valid) rows from the right for which their key matches an associated

$O.k$	$O.a$	$O.b$	$O.V$	$O.T_{id}$
-	-	-	0	-
k_1	$L.a_1$	0	1	0
k_2	$L.a_2$	0	1	0
k_2	0	$R.b_1$	1	1
k_2	0	$R.b_2$	1	1
k_2	0	$R.b_3$	1	1
k_3	0	$R.b_4$	1	1

unique key on the left:

$$\begin{aligned} O.V_i := O.V_i \wedge [O.T_{id,i} = 1] & \quad \text{valid row from } R \\ \wedge [\exists j : O.k_j = O.k_i & \quad \text{exists matching key} \\ \wedge O.T_{id,j} = 0 & \quad \text{from left} \\ \wedge O.V_j = 1] & \quad \text{which is valid} \end{aligned}$$

Due to previously having sorted on $V||k||T_{id}$, we know that if any such $O.k_j$ exists, it will occur as the first position of this k-group. If not, a row from the right appears as the first position of this k-group, and no rows with this key are in the output of the join. We use the DISTINCT operator to mark the first row of each group with a 1. Then, the join output is formed by updating a temporary valid bit, $O.V_o := O.V \wedge \neg\text{DISTINCT}(V, k)$. That is, $O.V_o$ invalidates any valid rows which are the *first* of their (V, k) group.[3]

For each group four possibilities exist, shown in the tables below.

Case I

$O.k$	¬DIST	$O.V$
...	...	0
...	...	...
$L.k$	0	1
$L.k$	1	1
$L.k$	1	1
$L.k$	1	1

Case II

$O.k$	¬DIST	$O.V$
...	...	0
...	...	...
$L.k$	0	1
$L.k$	1	1
$R.k$	1	1
$R.k$	1	1

Case III

$O.k$	¬DIST	$O.V$
...	...	0
...	...	...
$L.k$	0	1
$R.k$	1	1
$R.k$	1	1
$R.k$	1	1

Case IV

$O.k$	¬DIST	$O.V$
...	...	0
...	...	...
$R.k$	0	1
$R.k$	1	1
$R.k$	1	1
$R.k$	1	1

Case I There are duplicate keys on the left, and no matching keys on the right. The validity update procedure marks the first row from the left invalid, and aggregation (below)

[3]We cannot directly update $O.V$, because it will later be used as an aggregation key, during the execution of which we propagate changes to $O.V_o$. However, changing aggregation keys during execution breaks correctness, so we keep a temporary column.

will mark the entire group invalid. This represents (possibly duplicated) primary keys with no foreign key rows.

Case II There are duplicate rows on the left, and greater than zero matching keys on the right. The first left row is marked invalid, and aggregation similarly invalidates the entire left group. The right group remains valid.

Case III This is another matching case, but with exactly one primary key on the left. The single row from the left is invalidated, but the aggregation makes no changes. Case III is frequently observed in realistic workloads, where our join operator operates on an explicit primary-key foreign-key relation.

Case IV There are no matching rows on the left. Now we invalidate the *first row from the right*, and the subsequent aggregation invalidates *all rows on the right*. With no key on the left, this key is not be part of the output table.

In Cases I and II, there are multiple matching "primary keys"; we operate as if only the first existed. We tend to avoid this mode of operation, since it does not follow the semantics of pk-fk relations, but it is useful in some environments, such as when performing semi-join. It also allows us to natively perform `COUNT(DISTINCT(...))` operations over many-to-many joins (with duplicates on both sides).

After the invalid rows are marked, a final aggregation is applied (as part of the copy step, below) to similarly invalidate all rows in the group. This aggregation is a special case of the aggregation step outlined below, as the aggregation keys here are $V \mid\mid k \mid\mid T_{id}$. This allows for an invalidated left row to invalidate all other left rows, and an invalidated right row to invalidate all other right rows, but invalidation will never cross the table boundary.

The size of the output of a one-to-many inner join is bounded by the size of the right table: assume every key on the right is present exactly once on the left. Then the output of the join is exactly the right table. Because our protocol is oblivious, it must be input-independent, even in the worst case. Thus our output must always exactly the size of the right table; we use the valid column to mark rows not actually present in the join output.

The cardinality equivalence means it is most efficient to build the output of the join within the rows of the right table. As such, all data from the right is included in the output by default. Data from the left table, on the other hand, must be explicitly copied. This operation is performed with a *group-by aggregation* and implemented via AggNet. In practice, we have noticed that this default behavior also matches the semantics of many queries, so few `copy` aggregations are required.

Groups are defined as $V \mid\mid k$ (note the exclusion of the table ID column). The aggregation performed is a simple `Mux`, where the selection bit denotes whether values belong to the same group. We perform a reverse aggregation, which has the effect of aggregating *down*, and copying the first row of each multiplex group into all other rows of that group. Where a row on the left exists, it is the first row, due to the final sort by T_{id}; should multiple left rows exist, we only copy the value from the first. All left rows will be invalidated (Case II, above). The join operator does not support, at this stage, many-to-many-key joins. However, with a prior aggregation, or under certain conditions (e.g., the join performs a Distinct$(\cdot)$ aggregation), our operator still applies. See Section 6 and 6.3 for more details. □

Unifying join and aggregation. The validity-update procedure of join looks very similar to aggregations performed over joined tables. For improved efficiency, we would like

to combine these two operations into a single unified control flow, which will approximately halve the number of operations required.

Theorem 2 (Informal)**.** *Aggregations can be performed in the same control flow as the join operator provided the following conditions are met:*

1. *The aggregation keys are the same as the join keys.*
2. *No aggregation key is also an aggregation output.*
3. *No aggregation input is also an aggregation output for a different aggregation.*

The intuition behind this theorem is that the conditions preclude any interaction between individual aggregations, so they can be evaluated in parallel: calling AggNet$(f_1, \dots)$, AggNet$(f_2, \dots)$, AggNet$(f_3, \dots)$ is equivalent to

$$\text{AggNet}(\{f_1, f_2, f_3\}, \dots)$$

If any of the conditions are not true, these calls are *not* independent (and may have side effects), so sequential calls to the operator are required.

Handling other scenarios. Our operator only provides correct semantics for the case where there are at most unique keys on the left. To handle duplicate keys, we must first pre-aggregate over the left key. This requires an additional call to AggNet, after which the aggregated table has unique keys and aggregated values. Then, our operator can be applied. We note that this collapsing is only valid for self-decomposable aggregation functions.

This general technique is also applicable, as alluded to above, if one of the join or aggregation keys are a prefix of the other. (i.e., $K_j = K_a||K^*$ or $K_a = K_j||K^*$). Then, we sort on the longest combined key (argmax$(|K_j|, |K_a|)$) and perform the join normally. If the aggregation keys are a prefix of the join keys, we need to handle the possible interspersed rows from the left ($T_{id} = 0$): AggNet requires all rows in a group to be adjacent. We can accomplish this by masking rows from the left from an identity element for the given aggregation (e.g. for `sum`, 0; for `min`, ∞.) Then, while the join occurs over $K_j = K_a||K^*$, we can still aggregate over K_a by ignoring all K^* keys and taking advantage of the masked identity elements.

Finally, if the aggregation is applied to columns from both the left and the right (as in the Secure Yannakakis query, [WY21]) we must first break the aggregation function apart, apply either of the techniques above, and post-aggregate over the joined table. In the Secure Yannakakis example, we have:

```
SELECT T.class, SUM(S.cost * (1 - R.coinsurance))
FROM R, S, T
WHERE R.person=S.person AND S.disease=T.disease
GROUP BY T.class;
```

We observe that we can first pre-aggregate the quantity `1-R.coinsurance` per `person`, join $R \bowtie S$ on `person`, compute the product with `cost`, and then post-aggregate by summing over `disease` and `class`. This rewriting applies because the `SUM` function is self-decomposable.

Custom Aggregatons. Orq users can also define custom aggregation functions, as in the following example that computes a product:

```
template<typename A>
A prod(const A& a, const A& b) {
    return a * b;  // User-defined logic here
}
```

ORQ will call $\texttt{Mux}(g, a, \texttt{prod}(a, b))$ within the aggregation control flow, where `g` denotes whether the elements belong to the same group and `a`, `b` are slices of the vector to be aggregated. For each pair of elements in the same group ($g_i = 1$), the aggregation function `prod` is applied; otherwise, no update occurs. For the example above, this will result in obliviously multiplying all elements of each group.

Unique-key joins. An additional optimization is possible in the case of unique keys on both sides of the join. (We assume this information is included in the tables' public schema.) In this case, the call to AGGNET is unnecessary: since each row on the left has at most one match on the right, the call to DISTINCT is sufficient for computing the join; the output of is then bounded by $\min(|L|, |R|)$ rather than just $|R|$. Additionally, in this context, aggregating over the join keys is a nonsensical operation, so we omit it. This operation is now effectively just a PSI protocol; many more optimizations are possible. However, we only use this optimized join algorithm for the comparison with SecretFlow, since the only join implementation they provide is a PSI-join which requires unique keys.

D.1 Correctness of other types of joins

Outer joins. Outer joins work similarly to an inner join but with different validation rules. That is,

- A **left-outer join** is an inner join, plus all rows from the left.
- A **right-outer join** is an inner join, plus all rows from the right.
- A **full-outer join** contains all rows from both input tables, and is effectively just a concatenation with aggregation.

Full-outer join is the simplest case. Since we take all rows, there is nothing new to invalidate. We copy the valid bits from each input table and apply them to the joined output, which is just a concatenation $L||R$. (We may also sort, so that user-defined aggregations can be applied, if necessary.)

For a right-outer join, we do not invalidate any rows from the right. But by the semantics of our join operator, we only include new columns from the left if explicitly specified by the user with a `copy` aggregation. Any rows *originally* from the left table should always be removed. This validation rule, therefore, is also quite simple: $O.V \leftarrow O.V_{LR} \wedge O.T_{id}$, where rows from the left have $T_{id} = 0$.

To compute a left outer join, we can compute the inner join but should not invalidate any left-table rows. The desired relation is shown in Table 6. We invalidate rows from R which are the first of their group. When this occurs, it means no such row from L exists, so this row is not in the output relation. To only keep *unique* (rather than all) rows from L, the second clause could instead be replaced with $O.T_{id} \oplus \text{DISTINCT}(K)$.

Table 6: Left outer join validity-update rule $O.V \leftarrow O.V_{LR} \wedge \neg(O.T_{id} \wedge \text{DISTINCT}(K))$

V_{LR}	T_{id}	DISTINCT	V
0	X	X	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Semi-join. The semi-join $L \ltimes R$ returns all rows in L which match a row in R. Observe that we can reframe this operator into one we have already seen by swapping the tables: which operator returns all rows in R which match a row in L? This is the inner join $R \bowtie L$. So, to implement $L \ltimes R$, we simply call $R \bowtie L$, and then project the result columns to maintain the semantics of semi-join (namely, only return columns in L). This final operation requires no extra work because our inner-join operator, by default, returns all data in columns from the "right" (here, L). We acknowledge that this transformation of semi-join to inner join is not correct in general, but rather a byproduct of the way our oblivious control flow is implemented for one-to-many joins.

To see why this rewriting works, consider what rows are *removed* from $R \bowtie L$ by the validity-update procedure of our inner join protocol: all rows from R, and all rows from L which do not have a matching "primary key" in R. Of course, in a semi-join, table R is not part of the output relation, so we should always remove all rows from R. And, if a row in L does not have a match in R, then it should be removed from the output, by the definition of semi-join.

Anti-join. The inverse of semi-join is anti-join, $L \triangleright R$, which *removes* from L all rows matching one in R. We apply a similar transformation: we need an operator which will ignore all rows from R, and keep rows from L which do not match any in R. A partial answer is the *right-outer join* from above, again applied in the opposite direction, $R \bowtie\!\!\!\subset L$. This would keep all rows in the "right" (here, L), but it would also incorrectly keep rows from L which matched one from R.

The solution here is to apply a special aggregation over the valid bit[4] which copies the valid bit of the *first* row of a group to all other rows in the group. We do not include $O.T_{id}$ as an aggregation key. Thus, we invalidate all rows for which the first row in that group was previously invalidated by the (right outer) join. Since this join will invalidate all "left" ($= R$) rows, any group for which *a row exists in* R will have its *first* row invalidated by the join, and in the subsequent aggregation, all such rows from this group will be invalidated. This correctly implements anti-join; to match the semantics of the operator, we again project to the columns on the left. Thanks to our unified join-aggregation control flow, this validation update is performed at the same time as the join, and incurs the same cost as a copy aggregation (i.e., $O(n \log n)$ multiplex operations) in a normal join. Again, we acknowledge that this rewriting of anti-join to right-outer join is not correct in relational algebra, but a convenient optimization in our setting.

One side effect of this rewriting (for both semi-join and anti-join) is that our implementations of these operators transparently handles duplicates in R, by Cases I and II, above. Both semi-join and anti-join only consider the *existence*, in table R, of keys in L, so we need no special handling for duplicate keys.

D.2 Proof Sketch of AggNet correctness

In this section, we provide an argument for why the AggNet protocol achieves correctness. As an illustrative example, consider a table with $n = 2^\ell$ rows and three columns: a key column K, input column A, and output column G. We use these parameters for convenience but emphasize that the ideas generalize to any database. Additionally, while the aggregation network requires that the *full* table size is a power of two, each key can have arbitrary cardinality. If the number of input rows is not a power of two, we pad the table with dummy (invalid) rows and remove them after the protocol terminates.

The first step of the algorithm is to copy column A into column G (Line 1), since the aggregation function is applied in place. From this point, we only consider column G.

[4]Specifically, this is just a `copy` aggregation, which already has special handling in order to copy rows from the left to right.

Claim. *Values across distinct keys are never aggregated.*

Take an arbitrary row i and distance d. Assume the keys in rows $(i, i+d)$ do not match. Then $b \leftarrow 0$ (Line 4) and we update $G_{i+d} \leftarrow \texttt{Mux}(0, G_{i+d}, g) = G_{i+d}$ (Line 6), so this row is not modified. Therefore, we only aggregate rows with matching keys.

Claim. *The algorithm correctly aggregates groups.*

AggNet expects an input table sorted on the key column. This puts all rows of a given key-group next to each other. Assume row j is the final row of some arbitrary group k of size $m \leq n$. Then, we want to argue that the final value $G_j = f(G_{j-m}, \ldots, G_j)$.

In the final round of the protocol, G_j is updated with the value $n/2$ indices earlier (assume for the sake of argument that $m > n/2$), $G_j \leftarrow f(G_{j-n/2}, G_j)$. In the prior round, each of these two values were updated with their own value along with the values $n/4$ indices prior. The algorithm proceeds in this manner, taking the next power of two in each round. In the first round we have a distance $d = 1$ so adjacent elements are aggregated.

This procedure is easy to visualize if we consider this aggregation structure as a tree, where each "prior index" is a left branch, and the "same index" is a right branch. AggNet builds a binary tree, where the path through the tree from G_j to G_{j-h}, for arbitrary $h \leq m$, corresponds to the binary representation of h. Each leaf node has a single unique path to G_j, since each offset h has a unique binary representation. Thus, each input value is aggregated once and in order of its appearance in the original list. Since the aggregation function f is self-decomposable, we have that $f(f(X), f(Y)) = f(X, Y)$, so by induction we can show that the final result is $f(G_{j-m}, \ldots, G_j)$. □

D.3 Trimming the output of Join operations

In Section 6, we discussed bounding the outputs of joins to the size of the right table. Here, we elaborate on the heuristic used to automatically govern this trim operation. It is important that, due to the obliviousness of our operators, the "size" of tables discussed in this section refers to the secret-shared table size, *not* the number of valid rows (which is not known at runtime).

First, we provide intuition by way of an example. Consider a join between two asymmetric tables, $|L| = 10$ and $|R| = 10^6$. After join, we have a larger table of size $|O| = 10 + 10^6$. Clearly it is not worth trimming: we must perform valid-bit sort on a table of size $|O| \approx |R| = 10^6$, but are only able to remove $|L| = 10$ elements. However, if a join is more symmetric, say $|L| = 10^6$ and $|R| = 2 \cdot 10^6$, trimming is much more promising. While we sort on $|O| = 3 \cdot 10^6$ elements, we can then remove $1/3$ of the output rows.

Trimming brings significant improvements in composed queries where we also perform aggregations on the output of the join. Since the odd-even aggregation algorithm requires power-of-two-sized inputs, trimming frequently allows us to "drop" below the next power of two, leading to significant speedups in query execution. In the analysis below, we optimize for such composed operation, assuming a join will be followed by additional joins with other tables.

To establish the heuristic, we define the abstract costs of each of our major operators. Let $J(n)$, $V(n)$, $S(n)$, and $A(n)$ be the respective costs of join, valid-bit sort, full sort, and aggregation over n rows (we abstract away aggregation functions at this phase).

In this analysis, we will say we first join two tables of size L and R, with $N := L + R$ the combined size. A subsequent join is performed with a table of size T. Intuitively, we want to trim if the additional cost (valid-bit sorting over N rows) is less than the future savings. Expressed formally, when $J(N + T) > V(N) + J(R + T)$, we should trim.

First, we observe that joins are superlinear: $\forall n, m > 0 : J(n+m) \geq J(n) + J(m)$. This

follows from both the superlinearity of both sorting and aggregation.

$$\begin{aligned} J(N+T) &> J(L) + J(R+T) \\ J(L) + J(R+T) &> V(N) + J(R+T) \\ J(L) &> V(N) \implies \text{trim} \end{aligned}$$

That is, we should trim when the cost of performing a join over the trimmed rows would be more expensive than performing a valid-bit sort over the entire table.

We now decompose each operator:

- Valid-bit sorting is implemented with radixsort so is approximately linear, $V(n+m) \approx V(n) + V(m)$.
- Joins consist of a two single-bit sorts (valid column plus T_{id}), a regular sort, and an aggregation; $J(n) \approx 2V(n) + S(n) + A(n)$.

Define $\alpha := R/L$. For most applications involving unique-key joins, $\alpha > 1$. Then,

$$\begin{aligned} V(N) &< J(L) \implies \text{trim} \\ V(L) + V(R) &< 2V(L) + S(L) + A(L) \\ (\alpha - 1)V(L) &< S(L) + A(L) \end{aligned}$$

It is difficult to reason, in general, about the cost of aggregation, $A(\cdot)$, since this cost depends heavily on (possibly user-defined) aggregation functions. However, it can be bounded from below considering only the control flow of the aggregation operator itself. Let ω be the bitwidth of values in the table. Then both aggregation and quicksort perform at least $n \lg(n)$ comparisons, which, under MPC, each require $\log \omega$ operations. Valid-bit sorting, on the other hand, performs $6n$ ω-bit permutation operations, each of which we can approximate as costing N multiplications, where N is the number of parties required by the MPC protocol. Inserting these costs into the above, we see that:

$$\begin{aligned} (\alpha - 1)6NL &< 2L \lg(L) \lg(\omega) \implies \text{trim} \\ 3\alpha N &< \lg(L) \lg(\omega) \end{aligned}$$

For our common use cases, $\omega = 128$ bits (due to input padding) and $N \in \{2, 3, 4\}$. The table below gives some example sizes, and confirms our intuitions about only trimming when we can remove sufficiently many rows from the left table.

L	α for $N = 3$PC	Trim when $R < \ldots$
100	5.2	516
10k	10.3	103k
1M	15.5	15.5M
100M	20.7	2.07B

A similar analysis can be performed for radixsort, giving a slightly different heuristic which nonetheless makes similar concrete decisions on real queries.

E Bandwidth Measurements

In Table 7, we provide the bandwidth usage for each query used in the paper. Since TPC-H queries have a wide variety of input sizes (nearly an order-of-magnitude spread at a fixed Scale Factor), we report bandwidth usage in kilobytes per row. While these results have been collected at SF10, query execution scales only slightly superlinearly, so the values below provide a good bandwidth estimate across all query sizes reported in Orq.

Measurements are also normalized per party; some protocols are not perfectly symmetric, so we divide the *total communication* by the number of computing parties (2 in SH-DM, 3 in SH-HM, and 4 in Mal-HM, respectively). We observe that SH-DM has just under twice the communication complexity of SH-HM, and Mal-HM about three times. The fix for the attack by Brüggemann et al. [BS26] has minimal communication overhead (consisting of only $O(\log n)$ constant-bandwidth checks), so is not expected to noticeably change the results for Mal-HM. Results for other experiments (§8.3) are provided in Tables 9, 10, 8, and 11.

Table 7: Bandwidth (KB) per row and computing party, for all queries and protocols used in Orq.

Query	SH-DM	SH-HM	Mal-HM	Query	SH-DM	SH-HM	Mal-HM
TPC-H Q1	33.1	18.5	51.4	Aspirin	39.9	21.2	61.2
Q2	65.0	35.2	100.2	Comorbidity	42.5	23.5	65.8
Q3	42.2	23.6	65.4	Credit	26.6	14.0	40.6
Q4	44.1	24.3	68.5	SYan	42.2	23.9	65.6
Q5	122.5	66.4	188.7	Patients	22.3	12.0	34.5
Q6	0.2	0.3	0.6	Market Share	17.8	10.0	27.3
Q7	111.9	61.0	173.4	Password	26.9	14.8	40.8
Q8	120.2	65.5	186.6	C. Diff	27.4	15.0	41.6
Q9	113.4	62.5	176.1	Secrecy Q2	32.7	18.4	50.6
Q10	73.6	40.5	114.7				
Q11	41.8	23.4	65.7				
Q12	46.9	25.5	72.0				
Q13	45.6	25.6	70.4				
Q14	18.0	9.9	27.9				
Q15	54.1	30.0	83.6				
Q16	94.2	51.3	144.3				
Q17	53.0	29.4	82.8				
Q18	86.6	47.5	133.3				
Q19	21.2	11.5	32.8				
Q20	82.1	45.8	127.0				
Q21	160.2	87.0	245.8				
Q22	16.6	9.3	26.2				

Table 8: Total bandwidth (MB) for SecretFlow and Orq in the experiment of Fig. 10. Orq's optimized radixsort achieves between 1.34× and 1.79× lower bandwidth than the SecretFlow algorithms.

Input Size	SecretFlow: SBK	Orq: RS (64b)	SecretFlow: SBKvalid	Orq: RS (32b)
100k	921.4	517.6	282.0	210.0
1M	9 212.4	5 176.0	3 212.2	2 100.0
10M	92 118.9	51 760.0	37 568.5	21 000.0

Table 9: Bandwidth (KB) per row and party for the queries used in Fig. 9 (left). Secrecy's high bandwidth is due to its $O(n^2)$ join operator and $O(n \log^2 n)$ sorting algorithm (bitonic sort).

Query	Secrecy	Orq
TPCH-Q6	0.3	0.3
Password	30.9	10.7
Credit	39.2	9.3
Comorbidity	56.4	23.7
C. Diff	63.1	12.1
Aspirin	2 820	10.7
TPCH-Q4	4 825	16.5
TPCH-Q13	1 712	20.0

Table 10: Bandwidth (bytes) per row and party for the queries used in Fig. 9 (right). SecretFlow's low bandwidth is due to its join operator, which leaks matching rows to parties and allows them to perform most subsequent operations locally (i.e., without communication).

Query	SecretFlow	Orq
S1	<1	14
S2	<1	28
S3	88	7 677
S4	286	7 735
S5	2 835	29 834

Table 11: Total bandwidth (MB) for MP-SPDZ and Orq in the experiments of Fig. 11 (oblivious radixsort on 2^k rows). Orq's optimized radixsort achieves between 1.57× and 65.5× lower bandwidth than the MP-SPDZ algorithm. MP-SPDZ crashes at 2^{22} in SH-DM and runs out of memory at 2^{25} in SH-HM and 2^{20} in Mal-HM.

	SH-DM (2PC)		SH-HM (3PC)		Mal-HM (4PC)	
k	MP-SPDZ	Orq	MP-SPDZ	Orq	MP-SPDZ	Orq
10	165.3	5.3	18.7	11.9	272.0	40.3
11	363.7	10.6	37.4	23.9	600.4	80.6
12	793.4	21.2	74.8	47.8	1 315.3	161.3
13	1 719.0	42.4	149.6	95.6	2 860.5	322.7
14	3 702.2	84.8	299.2	191.2	6 182.2	645.5
15	7 933.0	169.6	598.4	382.4	13 288.1	1 291.0
16	16 922.9	339.2	1 196.7	764.9	28 424.6	2 582.1
17	35 959.8	678.4	2 393.5	1 529.8	60 547.2	5 164.2
18	76 147.6	1 356.8	4 787.1	3 059.7	128 492.2	10 328.4
19	160 751.0	2 713.7	9 574.2	6 119.4	271 780.4	20 656.9
20	338 414.0	5 427.4	19 148.4	12 238.9	(OOM)	
21	710 650.0	10 854.8	38 296.8	24 477.9		
22	(Crash)		76 593.9	48 955.9		
23			153 187.5	97 911.8		
24			306 375.0	195 823.6		
25			(OOM)			