Practical Two-Party Secret-Shared Joins for SQL Analytics

Joining two secret-shared tables with only two parties — and keeping the result secret-shared

Joint work with Srinivasan Raghuraman and Peter Rindal

Presented by Harshal Shah

Visa Research

Presented on 2026-October-27 at WPEC 2026

NIST Workshop on Privacy-Enhancing Cryptography 2026

What Alice and Bob Actually Hold

	Alice (P_0)		Bob (P_1)
X	Alice's share of X	⊕	Bob's share of X
Y	Alice's share of Y	⊕	Bob's share of Y

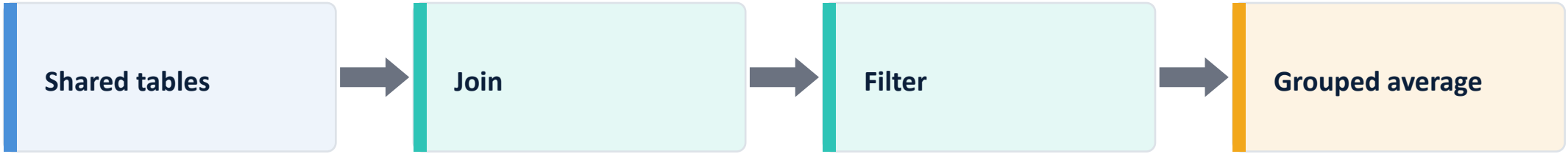
Reconstructing X or Y requires both halves — neither party's half reveals anything alone.

Output

Both parties hold shares of $X \bowtie Y$ (the join), never the plaintext tables.

This is NOT "Alice holds plaintext X, Bob holds plaintext Y." Both tables are jointly shared the whole time.

Why This Functionality Matters



Running example (illustrative): Customers ⋈ Transactions

Contribution

The first two-party protocols for secret-shared SQL-style joins (one-to-one and one-to-many) with secret-shared input and output, so the result can feed the next operation.

vs. standard Private Set Intersection (PSI)

PSI reveals the intersection in the clear — not composable with further secure computation.

What We Assume, Protect, and Support

Assume

Two parties

Semi-honest (passive) adversary

At most one corrupted party

Protect

Table contents stay secret-shared

All intermediate results stay secret-shared

Only row counts & schema are public

Support

One-to-one joins (unique keys, both sides)

One-to-many joins (duplicates on one side)

Composable with filters & aggregates

Public sizes & padding: row counts and the output table size are public by design; null/dummy rows pad the output so its size reveals nothing more.

The Operations the Protocols Need

Move / select shared rows

Rearrange rows into an input-dependent order without revealing the order.

Match keys privately

Decide which rows share a key — without opening either key.

Propagate / aggregate within groups

Copy or combine values across rows that share a key.

All three are built from one underlying tool →

Secret-Shared Permutations — the Engine Underneath

$\mathcal{F}$ below denotes an ideal functionality — a black-box helper protocol the main protocol calls.

$\mathcal{F}$_Select — one party knows the map

A plaintext mapping σ is held by one party and applied to shared data.

Fast. Used to build & query the cuckoo hash table in the one-to-one join.

$\mathcal{F}$_Perm — no one knows the map

The permutation itself is split: $\pi = \pi_1 \circ \pi_0$. Neither party alone knows π.

Needed whenever the reordering must stay hidden from both parties (e.g. sorting by a secret key).

Each individual share looks like an independent random permutation:

P_0 holds π_0

(looks random)

P_1 holds π_1

(looks random)

→

$\pi = \pi_1 \circ \pi_0$ applied to X

revealed to neither party

Enabling advance

Recent 2-party-efficient permutation protocols (Peceny, Raghuraman, Rindal, Shah — eprint 2024/547) are what make two-party joins practical.

Before: a 6-round oblivious switching network. Now: a single-round $\mathcal{F}$_Perm call.

$\mathcal{F}$_Sort doesn't sort — it outputs a secret-shared permutation, which $\mathcal{F}$_Perm then applies. Watch for sort → permute → process → unpermute in the one-to-many join.

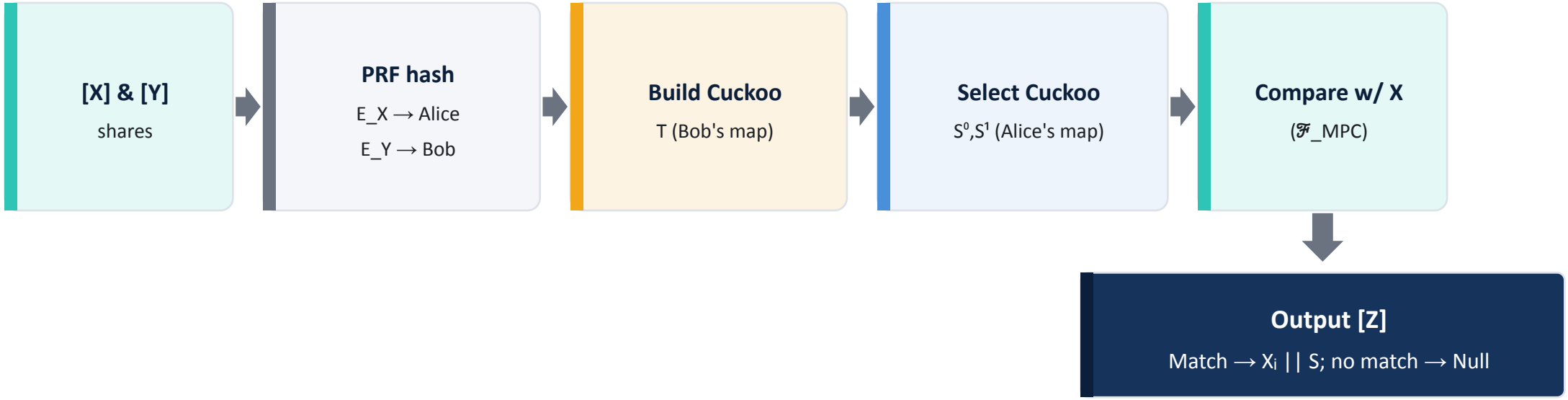
Walking Through a Tiny Unique-Key Example

Secret-shared

Visible to Alice (P_0)

Visible to Bob (P_1)

PRF = pseudorandom function • MPC = multi-party computation



Example: Customers (unique id) ⋈ Accounts (unique id) — each customer row meets at most one account row.

Introduce Duplicates into the Same Example

Customers (X) — unique key

cust_id	name
C1	Alice
C2	Bob

Accounts (Y) — duplicate key

cust_id	acc_id	amt
C1	A1	50
C1	A2	20
C2	A3	75

One row on the unique side (C1, Alice) must associate with multiple rows on the many side (A1 and A2)

Walking Through the Paper’s OM Construction

1. Sort keys

Combine key(X) // key(Y); $\mathcal{F}$_Sort returns a secret-shared permutation π.

2. Permute

$\mathcal{F}$_Perm applies π — X and Y rows interleave in sorted key order.

3. Control bits

An equality circuit flags adjacent rows that share a key.

4. Duplicate

An aggregation tree copies each X row forward into every matching Y row that follows.

5. Unpermute

$\mathcal{F}$_Perm applies π^{-1} to restore the original row order.

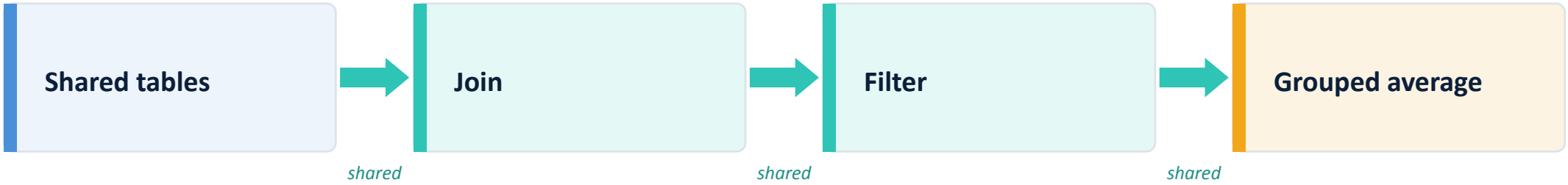
6. Combine

Output (X*, Y) for matches, Null otherwise.

This is the permutation pattern from the roadmap, now doing real work.

Before			→	After		
C1,Alice	A1,50 / A2,20		→	C1,Alice,A1,50	/	C1,Alice,A2,20

From Joins to Analytics



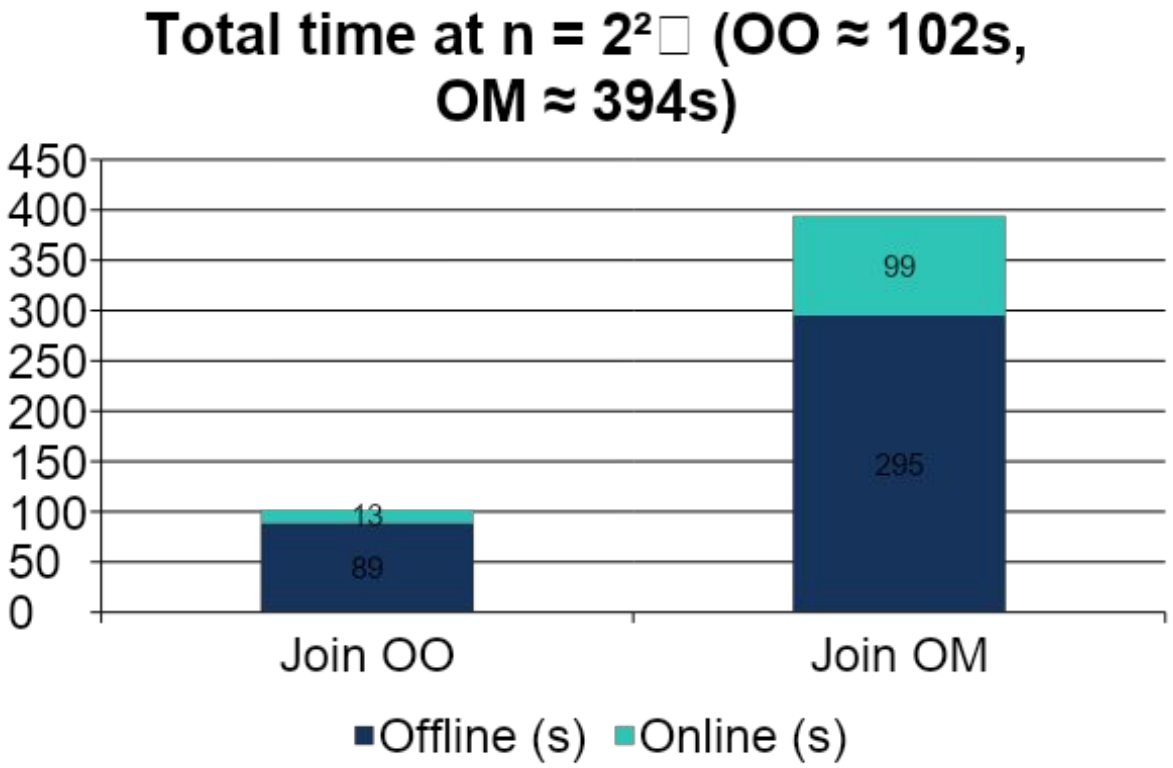
GroupBy + Average, explained once

Same backbone as the OM join: sort by group key ($\mathcal{F}$_Sort) → permute into group order ($\mathcal{F}$_Perm) → aggregate within each group with an associative operator ($\mathcal{F}$_Agg) → unpermute.

Also supported (briefly)

SUM, COUNT, MIN, MAX — WHERE / filter — UNION — left & full join. Same shared-in, shared-out interface each time.

Absolute Performance



Communication (Com, MB) at n=2^{20}

Join OO — offline 8,182 MB, online 5,219 MB

Join OM — offline 32,372 MB, online 21,633 MB

Simulated WAN (50ms, 1Gb/s)

Join OO total ≈ 310s — Join OM total ≈ 895s

(Table 2 of the paper)

Config: κ=128, λ=40, 32-bit join key, Apple M3 Pro, single thread. Left-hand numbers use a localhost 10+Gb/s link (Table 1); WAN box at right uses Table 2.

The Cost of No Honest Majority

Context, not a competitive claim

“Here is the performance cost of operating without an honest majority.”

The 3-party honest-majority protocol of [Badrinarayanan et al., CCS’22] is faster in total time, expected, since it assumes a trusted majority. Our online-phase time stays competitive with it up to $n=2^{16}$.

One verified engineering takeaway

4 threads ⟶ 2.5× speedup (measured). More threads didn’t help further — the workload was already saturated at four.

BACKUP

Benchmark Configuration and Full Tables

Protocol	n	Offline (s)	Offline (MB)	Online (s)	Online (MB)
Join OO	2^{12}	0.4	38	0.05	21
Join OO	2^{16}	5.5	517	0.5	326
Join OO	2^{20}	89	8,182	13	5,219
Join OM	2^{12}	1.4	141	0.1	82
Join OM	2^{16}	23	2,381	1.6	1,336
Join OM	2^{20}	295	32,372	99	21,633

Table 1 — localhost, 10+Gb/s, sub-ms latency.

Protocol	n	Offline (s)	Online (s)
Join OO	2^{20}	197	113
Join OM	2^{20}	540	355

Table 2 — simulated WAN, 50ms latency, 1Gb/s bandwidth. Both: κ=128, λ=40, 32-bit join key, Apple M3 Pro, single thread.

Complexity Accounting

Protocol	Time	Rounds
Join OO	O(n)	O(1) (constant)
Join OM	O(n log n)	O(log n)
GroupBy / Aggregate	O(n log n)	O(log n)

OM's O(n log n) time comes from the sorting step; its O(log n) rounds come from the aggregation-tree duplication step replacing a naive O(n)-round sequential copy.

Three Messages to Remember

1

Both the input and output tables are secret-shared at every stage.

2

The protocols cover one-to-one and one-to-many joins with only two parties.

3

Shared outputs enable subsequent secure analytics — filters, group-bys, aggregates — without ever opening the data.

Scope

Semi-honest security; one-to-one and one-to-many scope.

Repository QR code: include only once the public repository URL is confirmed.

Q&A

Thank you

Paper

Codebase