Efficient Oblivious Database Joins

Authors: Simeon Krastnikov, Florian Kerschbaum,
Douglas Stebila

Presenter: Xiling Li

May 21, 2025, Northwestern DB Reading Group

Outline:

- Motivation
- Contribution
- Threat Model
- Methods
- Experimental Results

Motivation

- Outsourced querying over cloud service providers (CSPs) is prevalent.
- However, CSPs are <u>untrusted</u>.
    - Adversary may have full control over CSPs.
- Goal: Enable to query over CSPs while revealing nothing about databases.

Obliviousness

- Beyond security guarantees (computing on encrypted data), memory access patterns during execution is a major source of information leakage.
- Authors mention standard O(n log n) sort-merge join as an example.
  - Adversary will learn data-dependent information!!!
- Oblivious RAM (ORAM) is a generic approach to protect the program oblivious.
  - However, it is asymptotically inefficient (O(log n) lower bound per access).
- Instead, this work discusses data-independent control flows and data padding.

Contribution

- This work focuses on oblivious binary equi-join operator.
  - Analogous to regular sort-merge join.

- With sorting network, the join is highly efficient, parallelizable and easy to verify.

- The proposed algorithm works for either machines with trusted hardware or circuit-based cryptographic protocols like secure multiparty computation or fully homomorphic encryption.

Table 1: Comparison of approaches for oblivious database joins. n_1 and n_2 are the input table sizes, $n = n_1 + n_2$, m is the output size, $m' = m + n_1 + n_2$, t is the amount of memory assumed to be oblivious. The time complexities are in terms of the number of database entries and assume use of a bitonic sorter for oblivious sorting (where applicable).

Algorithm/System	Time complexity	Local Memory	Assumptions/Limitations
Standard sort-merge join	$O(m' \log m')$	$O(1)$	not oblivious
Agrawal et al. [3] (Alg. 3)	$O(n_1 n_2)$	$O(1)$	insecure (see § 2.3.1 of [27])
Li and Chen [27] (Alg. A2)	$O(m n_1 n_2 / t)$	$O(t)$	–
Opaque [45] and ObliDB [13]	$O(n \log^2(n/t))$	$O(t)$	restricted to primary-foreign key joins
Oblivious Query Processing [5]	$O(m' \log^2 m')$	$O(\log m')$	missing details; performance concerns
Ours	$O(m' \log^2 m')$	$O(1)$	–

Computing on encrypted data

- Outsourced external memory
  - Client uses the server as external memory and compute locally after retrieving.
- Trusted execution environment
  - Code is executed within a secure enclave isolated from untrusted OS.
- Secure multiparty computation
  - It allows multiple parties to jointly compute a function over the secret input without revealing anything about the input.
- Fully homomorphic encryption
  - Computationally expensive for practical use due to one-time setup for circuits.

Threat Model

- The adversary has complete view of the public memory (e.g., RAM).
- System may has a limited local memory (e.g., in TEE) hidden from the adversary.
  - Adversary learns nothing except the runtime over the computation inside such memory.
- With either TEE or cryptographic protocols, the adversary cannot know the contents of the databases due to encryption.

Zooming into obliviousness

- Level I obliviousness
  - Public memory is oblivious while local memory with non-constant size uses for non-oblivious computation.
- Level II obliviousness
  - Both public/local memory are oblivious (doubly-obliviousness in Oblix).
- Level III obliviousness
  - Requires data-independent control flows.
- Revealing output length
  - Worst-case padding up to maximum possible size for output.

Table 2: Properties of three levels of obliviousness. Bottom portion of table shows vulnerability of programs satisfying these levels to timing (t), page access attacks on data (pd), page access attacks on code (pc), cache-timing (c), or branching (b) attacks when used in different settings.

Property/Setting	I	II	III
Constant local memory	×	✓	✓
Circuit-like	×	×	✓
Ext. Memory	t	t	✓
Secure Coprocessor	t	t	✓
TEE (enclave)	t, pd, pc, c, b	t, pc, c, b	✓
Secure Computation	n/a	n/a	✓
FHE	n/a	n/a	✓

Data-Independent Control Flow

$i \leftarrow 0$
while $i < secret$ **do**
  $i \leftarrow i + 1$

Loop

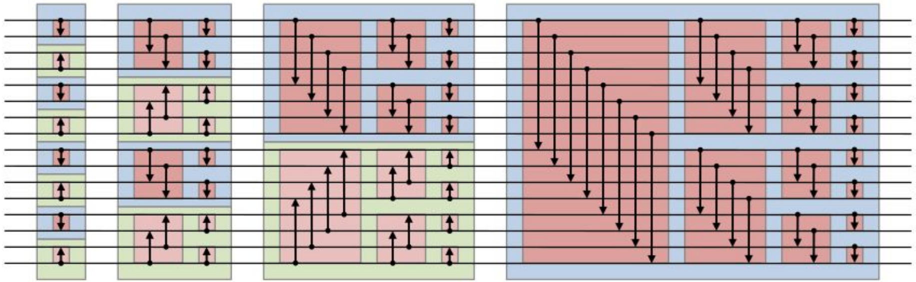
if $secret$ **then**
  $x_1 \leftarrow y_1$
  $x_3 \leftarrow y_3$
else
  $x_1 \leftarrow z_1$
  $x_2 \leftarrow z_2$

→

$$x_1 \leftarrow y_1 \cdot secret + z_1 \cdot (\neg secret)$$
$$x_2 \leftarrow z_2 \cdot 0 + z_2 \cdot (\neg secret)$$
$$x_3 \leftarrow y_3 \cdot secret + y_3 \cdot 0$$

Branching

- Transform from Level II to Level III.
- Loop depends on either a constant length or the input size.
  - Replace while loop with for loop.
- Eliminate the explicit branching
  - No if-else for encrypted data.
- Use bitonic sort for oblivious sort
  - Data-independent comp/swap.
  - Sort n entries in $O(n \log^2 n)$.



Bitonic Sort

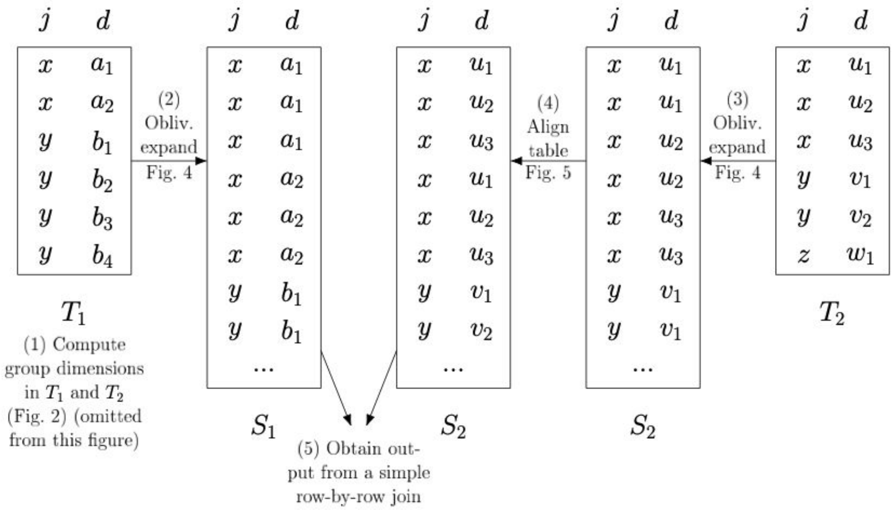
Overview

- Augment each table into join groups.
  - Resulting tables ordered by join attributes.

- Each row in both tables with same attributes is in the same group.
  - Obliviously expand each group to S1 and S2.
  - Group dimension for x is 6 due to 2 x rows in T1 and 3 x rows in T2 (Cartesian product).
  - |S1|=|S2|=sum of group dimensions.

- After expansion, aligns S2 with S1.
  - Previous 3 steps simulating the sort step in SMJ.

- Then, join each row in both S1 and S2 for result.
  - Simulate merge step in SMJ.
  - Unlike regular SMJ, it advances both cursors at the same time.



j is join attributes and d is other attributes

Algorithm 1 The full oblivious join algorithm

function OBLIVIOUS-JOIN($T_1(j,d), T_2(j,d)$)
  $T_1, T_2(j, d, \alpha_1, \alpha_2) \leftarrow$ AUGMENT-TABLES(T_1, T_2)
  $S_1(j, d, \alpha_1, \alpha_2) \leftarrow$ OBLIVIOUS-EXPAND(T_1, α_2)
  $S_2(j, d, \alpha_1, \alpha_2) \leftarrow$ OBLIVIOUS-EXPAND(T_2, α_1)
  $S_2 \leftarrow$ ALIGN-TABLE(S_2)
  initialize $T_D(d_1, d_2)$ of size $|S_1| = |S_2| = m$
  for $i \leftarrow 1 \ldots m$ **do**
    $T_D[i].d_1 \leftarrow S_1[i].d$
    $T_D[i].d_2 \leftarrow S_2[i].d$
  return T_D

Augmenting tables

- Union two input tables into Tc.
  - Distinguish each rows with table id (tid).
- Grouping Tc into contiguous blocks w.r.t. join attributes.
  - Obliviously sort on join attributes and tid.
- Fill dimensions for join groups in Tc for both input tables.
  - Two linear scan (one forward and one backward).
- Group Tc again to put rows in the same table contiguously.
  - Obliviously sort on tid, join attributes and data attributes.
- Split Tc into two augmented tables w.r.t. Tid.
  - We have augmented T1 and T2 respectively.
- Group dimension of each join key is alpha1*alpha2.

Algorithm 2 Augment the tables T_1 and T_2 with the dimensions α_1 and α_2 of each entry's corresponding group. The resulting tables are sorted lexicographically by (j, d). $n_1 = |T_1|$, $n_2 = |T_2|$, $n = n_1 + n_2$.

```
function AUGMENT-TABLES(T1, T2)                         ▷ O(n log^2 n)
    T_C(j, d, tid) ← (T1 × {tid = 1}) ∪ (T2 × {tid = 2})
    T_C ← BITONIC-SORT⟨j ↑, tid ↑⟩(T_C)
    T_C(j, d, tid, α1, α2) ← FILL-DIMENSIONS(T_C)
    T_C ← BITONIC-SORT⟨tid ↑, j ↑, d ↑⟩(T_C)
    T1(j, d, α1, α2) ← T_C[1 ... n1]
    T2(j, d, α1, α2) ← T_C[n1 + 1 ... n1 + n2]
    return T1, T2
```

(1) Downward scan: store incremental counts as temporary α_1 and α_2 attributes

j	d	tid	α_1	α_2
x	a_1	1	1	-
x	a_2	1	2	-
x	u_1	2	2	1
x	u_2	2	2	2
x	u_3	2	2	3
y	b_1	1	1	-
y	b_2	1	2	-
y	b_3	1	3	-
y	b_4	1	4	-
y	v_1	2	4	1
y	v_2	2	4	2
z	w_1	2	0	1

T_C

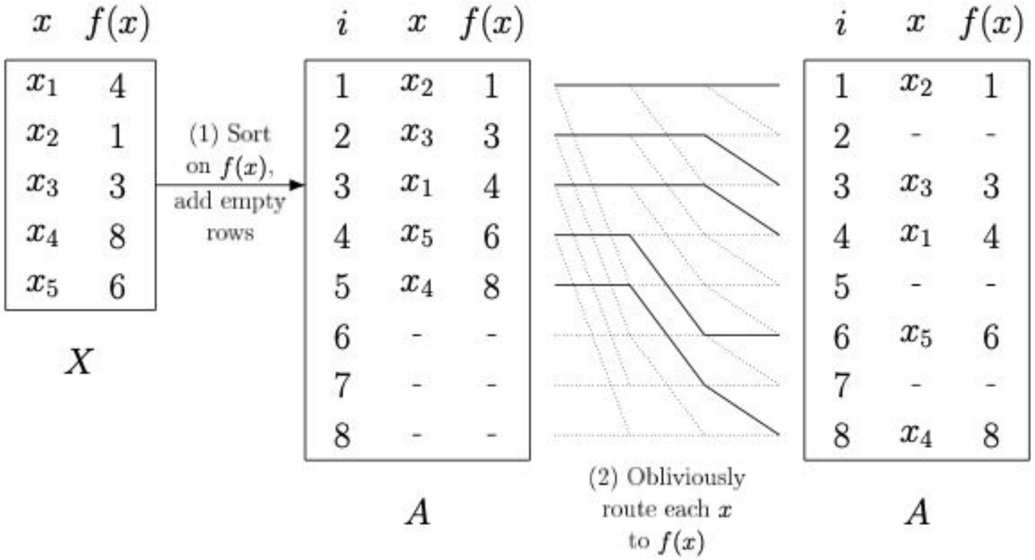
FILL-DIMENSIONS

(2) Upward scan: propagate correct α_1 and α_2 values stored in "boundary" entries

j	d	tid	α_1	α_2
x	a_1	1	2	3
x	a_2	1	2	3
x	u_1	2	2	3
x	u_2	2	2	3
x	u_3	2	2	3
y	b_1	1	4	2
y	b_2	1	4	2
y	b_3	1	4	2
y	b_4	1	4	2
y	v_1	2	4	2
y	v_2	2	4	2
z	w_1	2	0	1

T_C

Oblivious Distribution



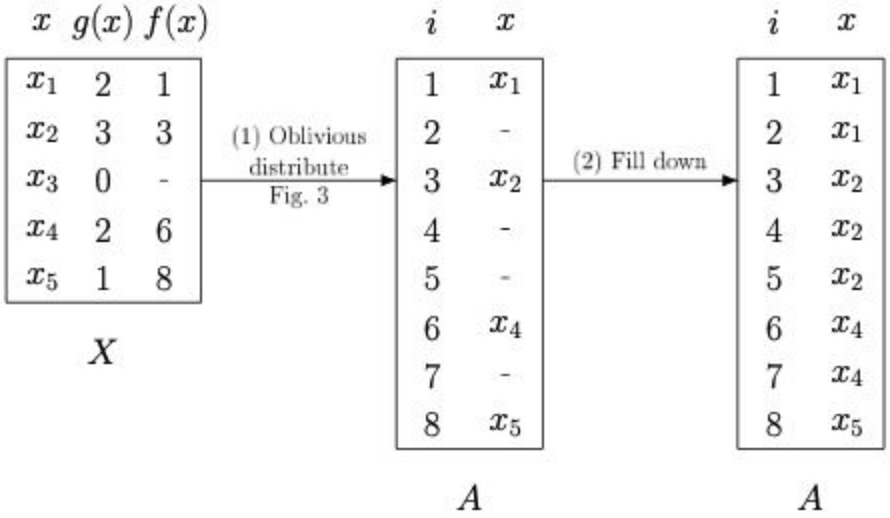
Algorithm 3 Obliviously map each $x \in X$ to index $f(x)$ of an array of size $m \geq n$, where $f : X \to \{1 \dots m\}$ is injective.

```
function OBLIVIOUS-DISTRIBUTE(X, f, m)
    A[1 ... n] ← X
    BITONIC-SORT⟨f ↑⟩(A)                          ▷ O(n log^2 n)
    A[n + 1 ... m] ← ∅ values
    extend f to f̂ such that f̂(∅) = 0
    j ← 2^(⌈log_2 m⌉−1)
    while j ≥ 1 do                                ▷ O(m log m)
        for i ← m − j ... 1 do
            y ←* A[i]
            y' ←* A[i + j]
            if f̂(y) ≥ i + j then
                A[i] ←* y'
                A[i + j] ←* y
            else
                A[i] ←* y
                A[i + j] ←* y'
        j ← j/2
    return A
```

y and y' are variables in the local memory, which means read from public memory to local memory for each comp/swap.

- Before obliviously expanding T1 and T2 into S1 and S2, this work obliviously distributes first occurrence of each join attribute into a correct destination position.
- The case of m=n is equivalent to oblivious sort while m > n is not trivial.
- Runtime complexity - $O(n \log^2 n + m \log m)$: $O(n \log^2 n)$ sort + $O(\log m)$ iteration + $O(m)$ inner loop.

Oblivious Expansion



Algorithm 4 Obliviously duplicate each $x \in X$ $g(x)$ times.

```
function OBLIVIOUS-EXPAND(X, g)
    ▷ obtain f values and distribute according to f
    s ← 1
    for i ← 1...n do                                   ▷ O(n)
        x ←* X[i]
        if g(x) = 0 then
            mark x as ∅
        else
            set f(x) = s
        s ← s + g(x)
        X[i] ←* x
    A ← EXT-OBLIVIOUS-DISTRIBUTE(X, f, s − 1)
    ▷ fill in missing entries
    px ← ∅
    for i ← 1...s − 1 do                               ▷ O(m)
        x ←* A[i]
        if x = ∅ then
            x ← px
        else
            px ← x
        A[i] ←* x
    return A

function EXT-OBLIVIOUS-DISTRIBUTE(X, f, m)
    A[1...n] ← X
    BITONIC-SORT⟨≠ ∅ ↑, f ↑⟩(A)                        ▷ O(n log² n)
    if m ≥ n then
        A[n + 1...m] ← ∅ values
    extend f to f̂ such that f̂(∅) = 0
    continue as in O(m log m) loop of Algorithm 3...
    return A[1...m]
```

- After obliviously distributing first occurrence of each join attribute, linear scan A and duplicate join key to the null slots.

Aligning table

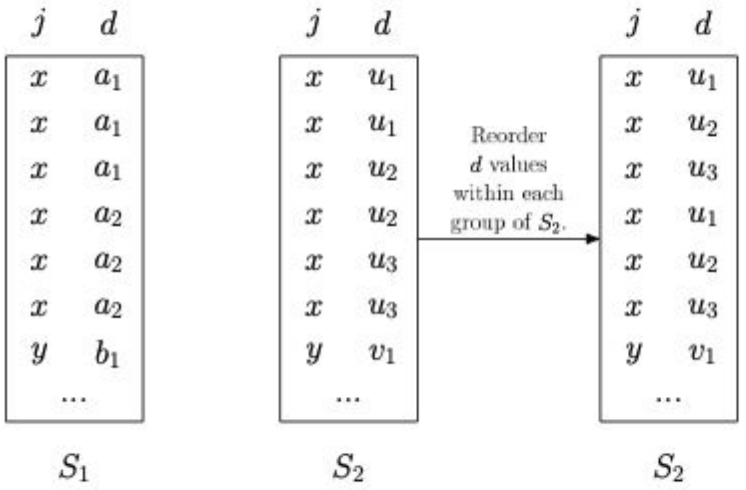
- Recall that S1 has alpha2 copies of T1 rows w.r.t. each join attribute.
    - E.g., 3 (x, a1) to match 3 distinct rows with x attribute in T2.

- Alignment matches each row in T1 to corresponding row in T2.
    - Match (x, u1), (x, u2) and (x, u3) once in T2 to (x,a1) and (x,a2) respectively.
    - Prevent duplicate matches for (x,a1) and (x, u1) and miss match for (x,a1) and (x,u3).

- Give each row in each group a correct ii and sort on (join attribute and ii) to put each rows in S2 into correct positions.
    - ii can split each dup rows into a certain distance.

Algorithm 5 Reorder S_2 so that its m entries align with those of S_1.

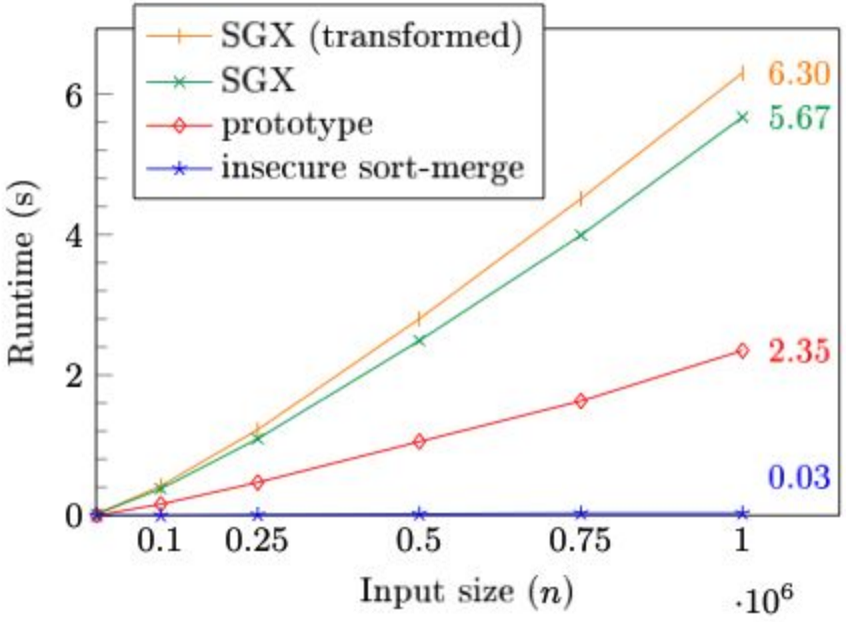
function ALIGN-TABLE(S_2)
  $S_2(j, d, \alpha_1, \alpha_2, ii) \leftarrow S_2 \times \{ii = \texttt{NULL}\}$
  for $i \leftarrow 1 \ldots |S_2|$ **do** $\quad \triangleright O(m)$
    $e \stackrel{\star}{\leftarrow} S_2[i]$
    $q \leftarrow$ (0-based) index of e within block for $e.j$
    $e.ii \leftarrow \lfloor q/e.\alpha_2 \rfloor + (q \mod e.\alpha_2) \cdot e.\alpha_1$
    $S_2[i] \stackrel{\star}{\leftarrow} e$
  $S_2 \leftarrow$ BITONIC-SORT$\langle j, ii \rangle(S_2)$ $\quad \triangleright O(m \log^2 m)$
  return S_2



Evaluation

Table 3: For each (non-linear) component of the algorithm: approximate counts of total comparisons (or swaps) when $m \approx n_1 = n_2$, as well as empirical share of total implementation runtime for $n = 10^6$.

Subroutine	Comparisons	Runtime
initial sorts on T_C	$n(\log_2 n)^2/2$	60%
o.d. on T_1, T_2 (sort)	$n_1(\log_2 n_1)^2/2$	25%
o.d. on T_1, T_2 (route)	$2m \log_2 m$	3%
align sort on S_2	$m(\log_2 m)^2/4$	12%
total (when $m \approx n_1 = n_2$)	$n(\log_2 n)^2 + n \log_2 n$	100%



- Implemented by C++ varying input size from n=10 to n10^6.
- Time complexity is O(n log^2 n + n log n).
- Memory is max(n1, m) + max(n2, m), since Tc in augmenting tables is the largest.

Future Work

- May consider complex queries including multi-way joins.
- Group-by aggregation may take fewer sorting steps.
- Extend the general-purpose primitives in this work like oblivious distribution and expansion to other oblivious algorithms in outsourced querying.

Thanks!