Towards Practical Oblivious Join

[Chang Xie Wang Li 22]

Michael Farber Brodsky

Preliminaries

- Oblivious RAM: A cryptographic object which can obfuscate memory access patterns with a small client memory requirement, and large memory which could be monitored: accessing encrypted data remotely while hiding access patterns.
- Memory access with “blocks” indexed by a position. Store and load operations are indistinguishable.
- ORAM is used as a black-box but Path-ORAM is specifically mentioned due to good performance and simplicity.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:
  - Which records are important and frequently used.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:
  - Which records are important and frequently used.
  - Which records are relevant for the current query.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:
  - Which records are important and frequently used.
  - Which records are relevant for the current query.
  - The whole structure of the B-tree, based on the way it is traversed.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:
  - Which records are important and frequently used.
  - Which records are relevant for the current query.
  - The whole structure of the B-tree, based on the way it is traversed.
  - Can potentially lead to understanding the order of records by each key.

Motivation: Memory access patterns

- Consider a normal database which is stored on an encrypted disk.
- The disk is evil! It records all the addresses we are accessing.
- It could probably find:
  - Which records are important and frequently used.
  - Which records are relevant for the current query.
  - The whole structure of the B-tree, based on the way it is traversed.
  - Can potentially lead to understanding the order of records by each key.

- Although the data in the records isn't revealed, this is a lot of information…

Problem definition

A way to use large public cloud storage without revealing anything about the data being handled in the database.

Problem definition

A way to use large public cloud storage without revealing anything about the data being handled in the database.

- Specifically, this paper presents several algorithms for oblivious joins on such a database.

Problem definition

A way to use large public cloud storage without revealing anything about the data being handled in the database.

- Specifically, this paper presents several algorithms for oblivious joins on such a database.
- An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can't be distinguished by anyone but the client.

Problem definition

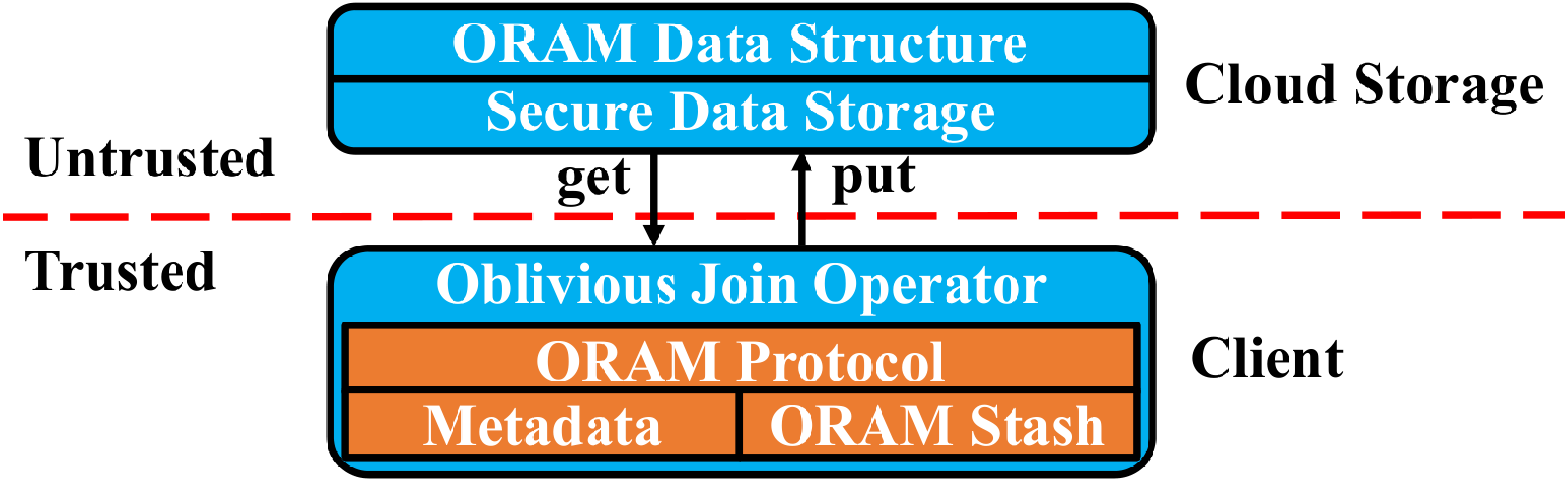


Figure 1: An overview of our oblivious join process.

Contributions

Contributions

- This paper solves:
  - Binary equi-join

    With two algorithms

Contributions

- This paper solves:
  - Binary equi-join

    With two algorithms
  - Binary band join

    Operators like *>, >=, <, <=* over the shared key

Contributions

- This paper solves:
  - Binary equi-join

    With two algorithms
  - Binary band join

    Operators like *>, >=, <, <=* over the shared key
  - *Acyclic* multiway equi-join

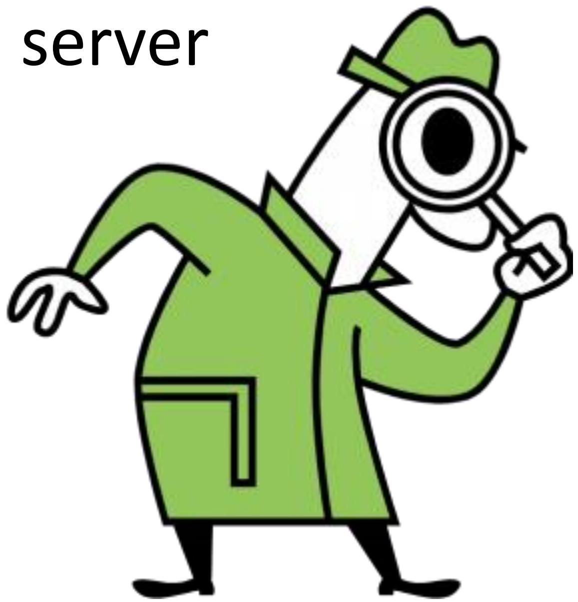
    We’ll see what Acyclic means later

Problem definition

- Security model: an “honest-but-curious” server.
- That means that the server will always act like an honest server, but might observe our memory access patterns as well as the data.

Problem definition

- Security model: an “honest-but-curious” server.
- That means that the server will always act like an honest server, but might observe our memory access patterns as well as the data.
- This is a useful security model because if the server gives malicious responses to a client which does store the database, the server provider would probably lose its reputation…



Database indexing

- In a normal database we usually have a B-tree index.

Database indexing

- In a normal database we usually have a B-tree index.
- We can do the same under oblivious RAM: the client stores the block ID of the root, and each node stores the block IDs of its children.

Database indexing

- In a normal database we usually have a B-tree index.
- We can do the same under oblivious RAM: the client stores the block ID of the root, and each node stores the block IDs of its children.
- The cost of this can be reduced if we separate each table's data and index into separate ORAMs. Furthermore, we could even use a separate ORAM for each level of a B-tree index.

Database indexing

- In a normal database we usually have a B-tree index.
- We can do the same under oblivious RAM: the client stores the block ID of the root, and each node stores the block IDs of its children.
- The cost of this can be reduced if we separate each table's data and index into separate ORAMs. Furthermore, we could even use a separate ORAM for each level of a B-tree index.
- Another approach is to integrate the B-tree information directly into the ORAM. When querying a node, we acquire the children simultaneously.

Oblivious binary join

- The paper presents two join implementations: <u>sort-merge</u> join and <u>index nested-loop</u> join.

Oblivious binary join

- The paper presents two join implementations: sort-merge join and index nested-loop join.
- These oblivious binary join implementations can’t be used for many-to-many join because that can leak information (the join degree).

Oblivious binary join

- The paper presents two join implementations: sort-merge join and index nested-loop join.
- These oblivious binary join implementations can't be used for many-to-many join because that can leak information (the join degree).
- The invariant kept for the implementation is that the two tables are accessed alternately, and then exactly one tuple is written, which may be a real or a dummy record.

Oblivious binary join

- The paper presents two join implementations: sort-merge join and index nested-loop join.
- These oblivious binary join implementations can't be used for many-to-many join because that can leak information (the join degree).
- The invariant kept for the implementation is that the two tables are accessed alternately, and then exactly one tuple is written, which may be a real or a dummy record.
- Finally, the dummy records are obliviously filtered out of the output table.

Why two algorithms

- Generally, sort-merge join and index-nested loop join each have their own advantages.

Why two algorithms

- Generally, sort-merge join and index-nested loop join each have their own advantages.
- Sort-merge join:
  - Assuming the join is over keys, the complexity is $O(N + M)$
  - Usually only used for equality

Why two algorithms

- Generally, sort-merge join and index-nested loop join each have their own advantages.
- Sort-merge join:
  - Assuming the join is over keys, the complexity is $O(N + M)$
  - Usually only used for equality
- Index-nested loop join:
  - Preferred if one of the sides of the join (N) has few rows, as the complexity is $O(N \log M)$
  - Works with operators which aren't equality

Why two algorithms

- Generally, <u>sort-merge join</u> and <u>index-nested loop join</u> each have their own advantages.
- Sort-merge join:
  - Assuming the join is over keys, the complexity is $O(N + M)$
  - Usually only used for equality
- Index-nested loop join:
  - Preferred if one of the sides of the join (N) has few rows, as the complexity is $O(N \log M)$
  - Works with operators which aren't equality
- Hash join is not relevant because hash join uses a large working memory, which is something we're trying to avoid.

A	B
1	1
2	1
2	2
3	1

Table T_1

$\bowtie$

A	C
1	1
2	1
2	2
2	3

Table T_2

A-to-A comparison

Join Comparison

T_1		T_2		res	getNext()
A	B	A	C		
1	1	1	1	$= 0$	
		2	1	< 0	$T_1 \rightarrow (2, 1)$
2	1	1	1	> 0	$T_2 \rightarrow (2, 1)$
2	1	2	1	$= 0$	
		2	2		
		2	3		
		$\perp$		< 0	$T_1 \rightarrow (2, 2)$
2	2	2	1	$= 0$	

Reached end

T_1		T_2		res	getNext()
A	B	A	C		
2	2	2	2	$= 0$	
		2	3		
		$\perp$		< 0	$T_1 \rightarrow (3, 1)$
3	1	2	1	> 0	$T_2 \rightarrow (2, 2)$
		2	2		$T_2 \rightarrow (2, 3)$
		2	3		$T_2 \rightarrow \perp$
		$\perp$		< 0	$T_1 \rightarrow \perp$
$\perp$		$\perp$		end	

Figure 3: An example of oblivious sort-merge equi-join.

The algorithm

(let’s not get too deep into it)

Match? Advance T_2 until different.
Then return to the beginning of this key in T_2 and advance T_1 once.

No match? Keep advancing whichever one is smaller, until we find a match.

Algorithm 1: Oblivious Binary Sort-Merge Equi-Join

Require: Input: two tables T_1 and T_2.
Output: join result table $T_{out} = T_1 \bowtie T_2$.

Initialize $T_{out} := \emptyset$.
Initialize $\text{tuple}[1,2] := \emptyset$.
for $i := 1$ to 2 **do**
  $\text{tuple}[i] := T_i.\text{getFirst}();$
while $\text{tuple}[1] \neq \perp$ **or** $\text{tuple}[2] \neq \perp$ **do**
  $\text{res} := \text{compare}(\text{tuple}[1], \text{tuple}[2]);$
  if $\text{res} = 0$ **then**
    $\text{begin} := \text{tuple}[2];$
    while $\text{res} = 0$ **do**
      $T_{out}.\text{put}(\text{tuple}[1] \bowtie \text{tuple}[2]);$
      $T_1.\text{getDummy}();\ \text{tuple}[2] := T_2.\text{getNext}();$
      $\text{res} := \text{compare}(\text{tuple}[1], \text{tuple}[2]);$
    $T_{out}.\text{put}(\perp);$
    $\text{tuple}[2] := \text{begin};$
    $\text{tuple}[1] := T_1.\text{getNext}();\ T_2.\text{getDummy}();$
  else
    $T_{out}.\text{put}(\perp);$
    if $\text{res} < 0$ **then**
      $\text{tuple}[1] := T_1.\text{getNext}();\ T_2.\text{getDummy}();$
    else
      $T_1.\text{getDummy}();\ \text{tuple}[2] := T_2.\text{getNext}();$
Obliviously filter out dummy records from T_{out}.
return T_{out};

What this algorithm achieves

- Remember the definition:

An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can’t be distinguished by anyone but the client.

What this algorithm achieves

- Remember the definition:

An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can't be distinguished by anyone but the client.

- This algorithm does NOT hide the size of the join – basically, if the join result is larger because a key is repeated more times, it's not hidden.

What this algorithm achieves

- Remember the definition:

An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can't be distinguished by anyone but the client.

- This algorithm does NOT hide the size of the join – basically, if the join result is larger because a key is repeated more times, it's not hidden.
- An adversary which can insert could be able to find the number of appearances of a key from the change in the amount of memory accesses.

What this algorithm achieves

- Remember the definition:

An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can't be distinguished by anyone but the client.

- This algorithm does NOT hide the size of the join – basically, if the join result is larger because a key is repeated more times, it's not hidden.
- An adversary which can insert could be able to find the number of appearances of a key from the change in the amount of memory accesses.
- Otherwise, the number of accesses, as well as the accessed tables if they're in separate ORAMs, is determined by the join's size.

What this algorithm achieves

- Remember the definition:

An oblivious join algorithm ensures that for databases of the same size and schema and joins over them with the same input/output size, the memory access traces have the same length and can't be distinguished by anyone but the client.

- This algorithm does NOT hide the size of the join – basically, if the join result is larger because a key is repeated more times, it's not hidden.
- An adversary which can insert could be able to find the number of appearances of a key from the change in the amount of memory accesses.
- Otherwise, the number of accesses, as well as the accessed tables if they're in separate ORAMs, is determined by the join's size.
- This can be mitigated with padding (extra accesses): e.g. to next power of 2

Oblivious index nested-loop binary equi-join

- Similarly, the two tables are accessed alternatively, and dummy records are used. Other than that, normal index nested-loop join.

Algorithm 2: Oblivious Index Nested-Loop Binary Equi-Join

Require: Input: two tables T_1 and T_2.
Output: join result table $T_{\text{out}} = T_1 \bowtie T_2$.

Initialize $T_{\text{out}} := \emptyset$.
Initialize $\text{tuple}[1, 2] := \emptyset$.
for $i := 1$ to $|T_1|$ **do**
    $\text{tuple}[1] := T_1.\text{getNext}()$;
    $\text{tuple}[2] := T_2.\text{getFirst}(\text{tuple}[1].\text{key})$;
    while $\text{match}(\text{tuple}[1], \text{tuple}[2]) =$ **true do**
        $T_{\text{out}}.\text{put}(\text{tuple}[1] \bowtie \text{tuple}[2])$;
        $T_1.\text{getDummy}()$;
        $\text{tuple}[2] := T_2.\text{getNext}()$;
    $T_{\text{out}}.\text{put}(\perp)$;
Obliviously filter out dummy records from T_{out}.
return T_{out};

Oblivious index nested-loop binary equi-join

- Similarly, the two tables are accessed alternatively, and dummy records are used. Other than that, normal index nested-loop join.

Algorithm 2: Oblivious Index Nested-Loop Binary Equi-Join

Require: Input: two tables T_1 and T_2.
Output: join result table $T_{\text{out}} = T_1 \bowtie T_2$.

Initialize $T_{\text{out}} := \emptyset$.
Initialize $\text{tuple}[1, 2] := \emptyset$.
for $i := 1$ to $|T_1|$ **do**
  $\text{tuple}[1] := T_1.\text{getNext}()$;
  $\text{tuple}[2] := T_2.\text{getFirst}(\text{tuple}[1].\text{key})$;
  while $\text{match}(\text{tuple}[1], \text{tuple}[2]) =$ **true do**
    $T_{\text{out}}.\text{put}(\text{tuple}[1] \bowtie \text{tuple}[2])$;
    $T_1.\text{getDummy}()$;
    $\text{tuple}[2] := T_2.\text{getNext}()$;
  $T_{\text{out}}.\text{put}(\perp)$;
Obliviously filter out dummy records from T_{out}.
return T_{out};

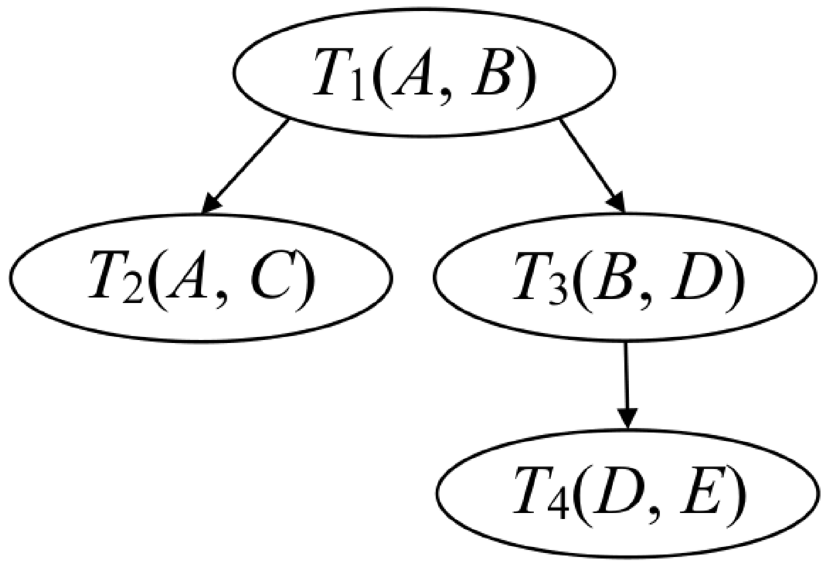
This can be extended for band-join (e.g. >, >=, <, <=), retrieving tuples until they don't match

Oblivious acyclic multiway equi-join

- By extending a binary equi-join algorithm to multiple tables (one join at a time into intermediate tables) we leak the intermediate table sizes – which is more information than just the join size.

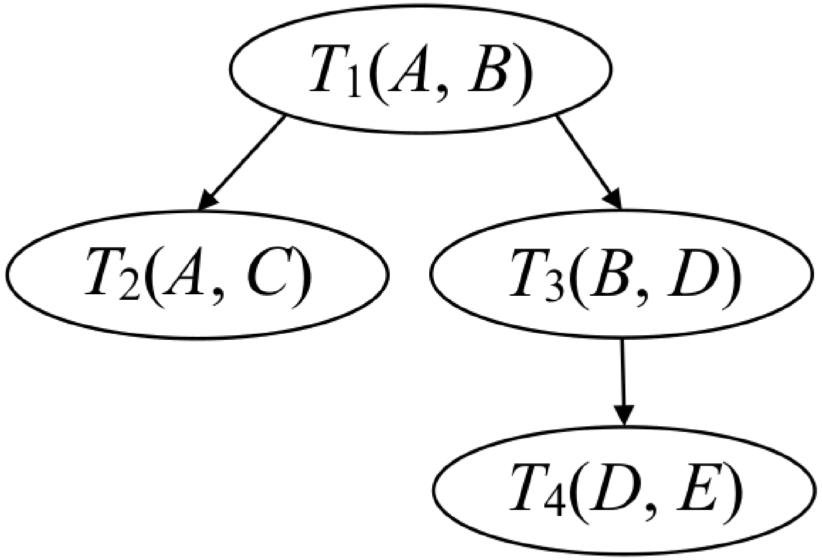
Oblivious acyclic multiway equi-join

- By extending a binary equi-join algorithm to multiple tables (one join at a time into intermediate tables) we leak the intermediate table sizes – which is more information than just the join size.
- The proposed join algorithm is an extension of the index nested-loop binary equi-join, and supports *acyclic multi-way equi-joins*.



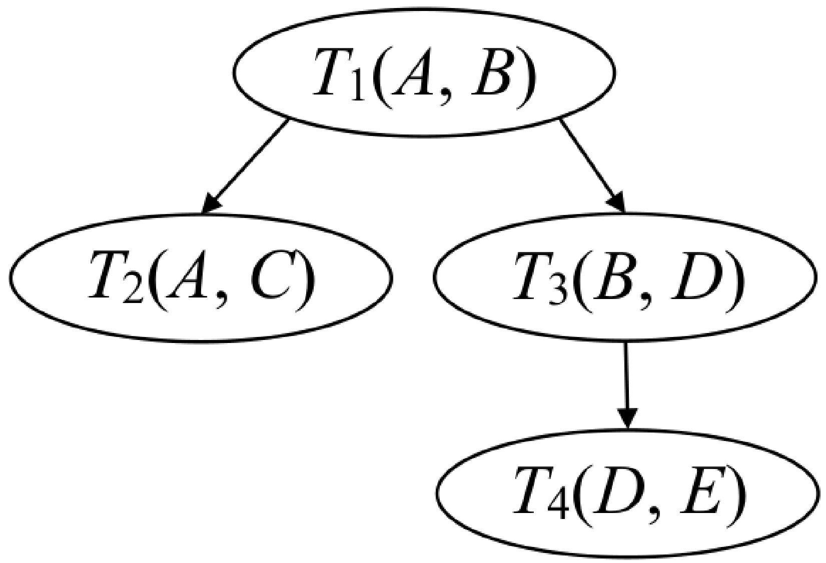
Oblivious acyclic multiway equi-join

- By extending a binary equi-join algorithm to multiple tables (one join at a time into intermediate tables) we leak the intermediate table sizes – which is more information than just the join size.
- The proposed join algorithm is an extension of the index nested-loop binary equi-join, and supports *acyclic multi-way equi-joins*.
- The algorithm iterates over T_1 and searches matched tuples in T_2 and T_3.



Oblivious acyclic multiway equi-join

- By extending a binary equi-join algorithm to multiple tables (one join at a time into intermediate tables) we leak the intermediate table sizes – which is more information than just the join size.
- The proposed join algorithm is an extension of the index nested-loop binary equi-join, and supports *acyclic multi-way equi-joins*.
- The algorithm iterates over T_1 and searches matched tuples in T_2 and T_3.
- Like previous oblivious join algorithms, it accesses each input table in round-robin.



Oblivious acyclic multiway equi-join

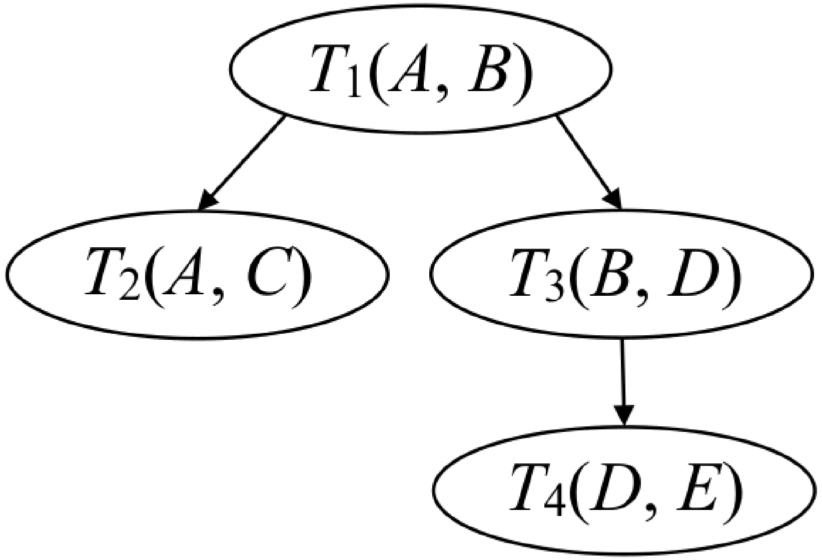
- By extending a binary equi-join algorithm to multiple tables (one join at a time into intermediate tables) we leak the intermediate table sizes – which is more information than just the join size.
- The proposed join algorithm is an extension of the index nested-loop binary equi-join, and supports *acyclic multi-way equi-joins*.
- The algorithm iterates over T_1 and searches matched tuples in T_2 and T_3.
- Like previous oblivious join algorithms, it accesses each input table in round-robin.
- Additionally, the number of join steps is padded to an upper bound for obliviousness.

The join result size

$$|T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\text{real}}|.$$

Oblivious acyclic multiway equi-join

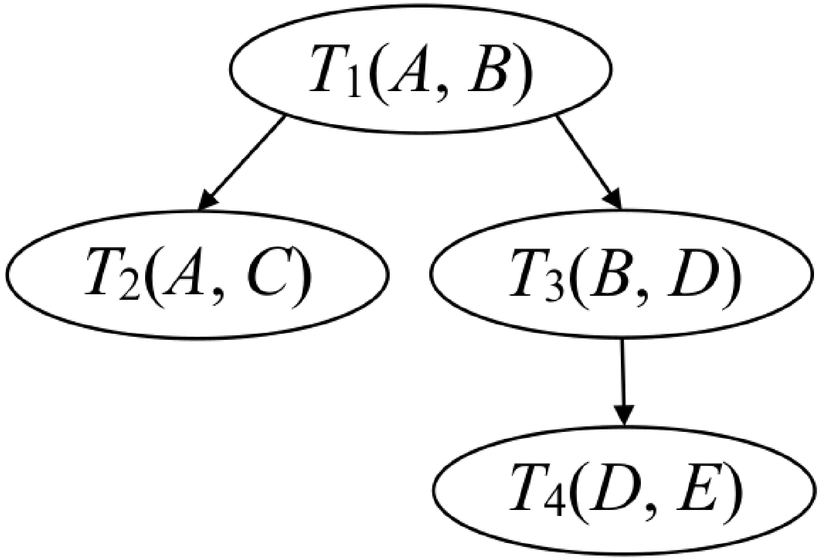
- The acyclicity becomes interesting to make this join more efficient, by using to the following observations:



Oblivious acyclic multiway equi-join

- The acyclicity becomes interesting to make this join more efficient, by using to the following observations:

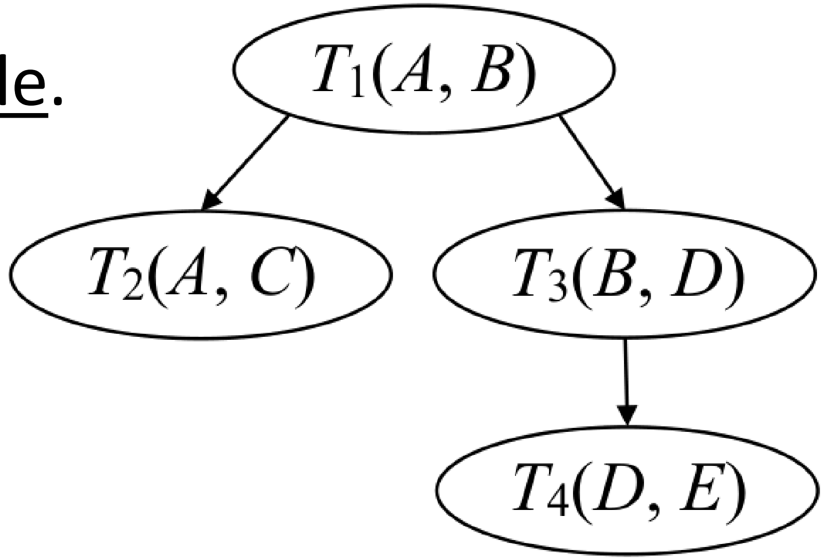
1. For any non-root table, if there is no match – then the parent's current key makes no contribution to the final join result, and can be safely disabled in the B-tree. This disabling is marked on leaves and can propagate up to the B-tree parents.



Oblivious acyclic multiway equi-join

- The acyclicity becomes interesting to make this join more efficient, by using to the following observations:

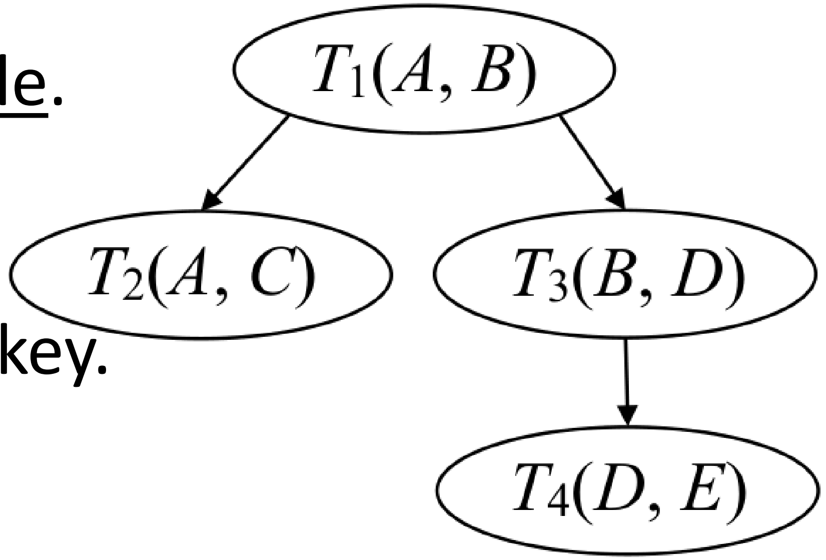
1. For any non-root table, if there is no match – then the parent's current key makes no contribution to the final join result, and can be safely disabled in the B-tree. This disabling is marked on leaves and can propagate up to the B-tree parents.
2. This can propagate upwards to the parent table.



Oblivious acyclic multiway equi-join

- The acyclicity becomes interesting to make this join more efficient, by using to the following observations:

1. For any non-root table, if <u>there is no match</u> – then the parent's current key makes no contribution to the final join result, and can be <u>safely disabled</u> in the B-tree. This disabling is marked on leaves and can propagate up to the B-tree parents.
2. This can propagate upwards <u>to the parent table</u>.
3. In equi-joins if the current tuple matches, but the succeeding tuple has a different key, there will be no more matches for the parent key. This can be marked in the B-tree leaves.



Oblivious acyclic multiway equi-join

Theorem: In that way, we can achieve the following bound for the number of steps (each step accesses the ORAM in all tables):

The join result size

$$|T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\text{real}}|$$

- General idea: If a tuple is enabled after being processed at least once, it will be enabled all the time. And after each tuple retrieval from each table, we will output exactly one (real or dummy) join record.

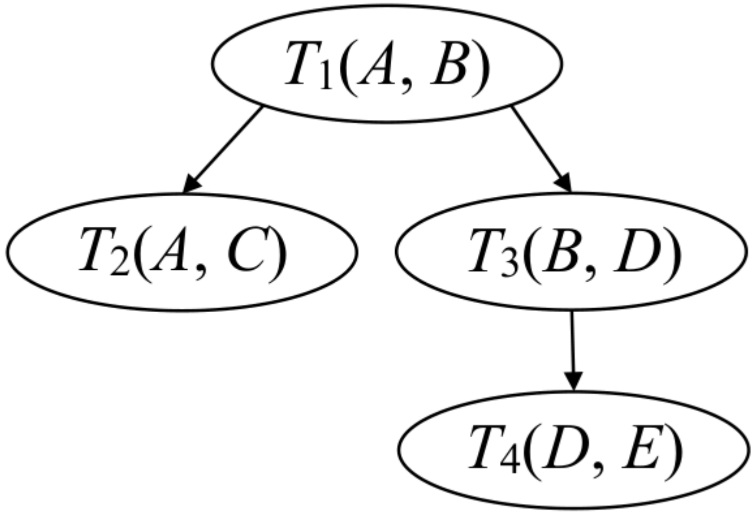


Table T_1

A	B
1	1
2	1
2	2
2	3

Table T_2

A	C
1	1
2	1
2	2
3	1

$\bowtie$

Table T_3

B	D
~~1~~	~~4~~
2	1
~~2~~	~~3~~

Table T_4

D	E
1	2
2	1
2	3

Join Comparison

T_1		T_2		T_3		T_4		match
A	B	A	C	B	D	D	E	
1	1	1	1	1	4	$\perp$		false
$\perp$		$\perp$		~~1~~	~~4~~	$\perp$		disable
2	1	2	1	2	1	$\perp$		false
2	2	2	1	2	1	1	2	true
2	2	2	1	2	3	$\perp$		false
$\perp$		$\perp$		~~2~~	~~3~~	$\perp$		disable
2	2	2	2	2	1	1	2	true
2	3	2	1	$\perp$		$\perp$		false
padding dummy operations								

Figure 6: An example of oblivious multiway equi-join.

Algorithm overview

Loop over T_1.
Join is a recursive function, and table numbers for parents are always less than for their children.

Algorithm 5: Oblivious Index Nested-Loop Multiway Equi-Join

Require: Input: ℓ tables $T_1, \cdots, T_\ell$.
Output: join result table $T_{\text{out}} = T_1 \bowtie \cdots \bowtie T_\ell$.

Initialize $T_{\text{out}} := \emptyset$.
Initialize $\text{tuple}[1 \dots \ell] := \emptyset$.
for $i := 1$ to $|T_1|$ **do**
    $\text{tuple}[1] := T_1.\text{getNext}()$;
    JOIN(2);
end for
Pad tuple retrievals and dummy output records to an upper bound.
Obliviously filter out dummy records from T_{out}.
Go over all index blocks and reset boolean tags in each entry.
return T_{out};

If there's no match for the parent's value, fail

If $j \neq$ index, then it must have failed, but I am not the parent which should get disabled. Backtrack to it…

If I am the target, disable me

Loop while there's more tuples matching the parent

Return a flag – if it is false, the parent should disable its tuple

```
function JOIN(j)
res := false;
tuple[j] := RETRIEVEFIRSTTUPLE(j, tuple[p(j)].key);
if match(tuple[p(j)], tuple[j]) = false then
    OUTPUTDUMMYRECORD(j);
else
    while true do
        if j = ℓ then
            res := true;
            OUTPUTREALJOINRECORD();
        else
            {flag, index} := JOIN(j + 1);
            if j ≠ index then
                return {flag, index};
            end if
            if flag = false then
                DISABLECURRENTTUPLE(j);
            else
                res := true;
            end if
        end if
        if HASNEXTMATCHEDTUPLE(j) = true then
            tuple[j] := RETRIEVENEXTTUPLE(j);
        else
            break;
        end if
    end while
end if
if res = false then
    return {false, p(j)};
else
    return {true, j − 1};
end if
end function
```

If there's no match for the parent's value, fail

If $j \neq$ index, then it must have failed, but I am not the parent which should get disabled. Backtrack to it…

If I am the target, disable me

Loop while there's more tuples matching the parent

Return a flag – if it is false, the parent should disable its tuple

```
function JOIN(j)
res := false;
tuple[j] := RETRIEVEFIRSTTUPLE(j, tuple[p(j)].key);
if match(tuple[p(j)], tuple[j]) = false then
    OUTPUTDUMMYRECORD(j);
else
    while true do
        if j = ℓ then
            res := true;
            OUTPUTREALJOINRECORD();
        else
            {flag, index} := JOIN(j + 1);
            if j ≠ index then
                return {flag, index};
            end if
            if flag = false then
                DISABLECURRENTTUPLE(j);
            else
                res := true;
            end if
        end if
        if HASNEXTMATCHEDTUPLE(j) = true then
            tuple[j] := RETRIEVENEXTTUPLE(j);
        else
            break;
        end if
    end while
end if
if res = false then
    return {false, p(j)};
else
    return {true, j − 1};
end if
end function
```

If there’s no match for the parent’s value, fail

If $j \neq$ index, then it must have failed, but I am not the parent which should get disabled. Backtrack to it…

If I am the target, disable me

Loop while there’s more tuples matching the parent

Return a flag – if it is false, the parent should disable its tuple

```
function JOIN(j)
res := false;
tuple[j] := RETRIEVEFIRSTTUPLE(j, tuple[p(j)].key);
if match(tuple[p(j)], tuple[j]) = false then
   OUTPUTDUMMYRECORD(j);
else
   while true do
      if j = ℓ then
         res := true;
         OUTPUTREALJOINRECORD();
      else
         {flag, index} := JOIN(j + 1);
         if j ≠ index then
            return {flag, index};
         end if
         if flag = false then
            DISABLECURRENTTUPLE(j);
         else
            res := true;
         end if
      end if
      if HASNEXTMATCHEDTUPLE(j) = true then
         tuple[j] := RETRIEVENEXTTUPLE(j);
      else
         break;
      end if
   end while
end if
if res = false then
   return {false, p(j)};
else
   return {true, j − 1};
end if
end function
```

If there's no match for the parent's value, fail

If $j \neq$ index, then it must have failed, but I am not the parent which should get disabled. Backtrack to it…

If I am the target, disable me

Loop while there's more tuples matching the parent

Return a flag – if it is false, the parent should disable its tuple

```
function JOIN(j)
res := false;
tuple[j] := RETRIEVEFIRSTTUPLE(j, tuple[p(j)].key);
if match(tuple[p(j)], tuple[j]) = false then
    OUTPUTDUMMYRECORD(j);
else
    while true do
        if j = ℓ then
            res := true;
            OUTPUTREALJOINRECORD();
        else
            {flag, index} := JOIN(j + 1);
            if j ≠ index then
                return {flag, index};
            end if
            if flag = false then
                DISABLECURRENTTUPLE(j);
            else
                res := true;
            end if
        end if
        if HASNEXTMATCHEDTUPLE(j) = true then
            tuple[j] := RETRIEVENEXTTUPLE(j);
        else
            break;
        end if
    end while
end if
if res = false then
    return {false, p(j)};
else
    return {true, j − 1};
end if
end function
```

If there's no match for the parent's value, fail

If $j \neq$ index, then it must have failed, but I am not the parent which should get disabled. Backtrack to it…

If I am the target, disable me

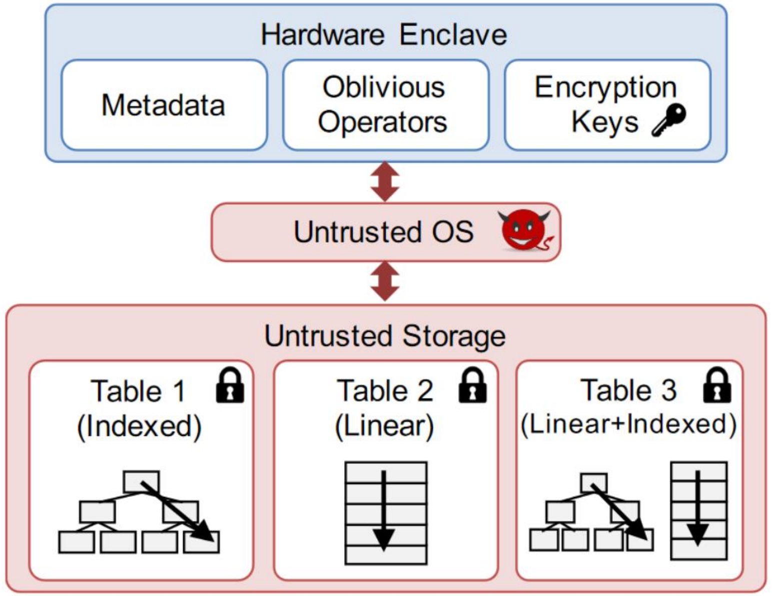
Loop while there's more tuples matching the parent

Return a flag – if it is false, the parent should disable its tuple

```
function JOIN(j)
res := false;
tuple[j] := RETRIEVEFIRSTTUPLE(j, tuple[p(j)].key);
if match(tuple[p(j)], tuple[j]) = false then
   OUTPUTDUMMYRECORD(j);
else
   while true do
      if j = ℓ then
         res := true;
         OUTPUTREALJOINRECORD();
      else
         {flag, index} := JOIN(j + 1);
         if j ≠ index then
            return {flag, index};
         end if
         if flag = false then
            DISABLECURRENTTUPLE(j);
         else
            res := true;
         end if
      end if
      if HASNEXTMATCHEDTUPLE(j) = true then
         tuple[j] := RETRIEVENEXTTUPLE(j);
      else
         break;
      end if
   end while
end if
if res = false then
   return {false, p(j)};
else
   return {true, j − 1};
end if
end function
```

Non-ORAM approaches

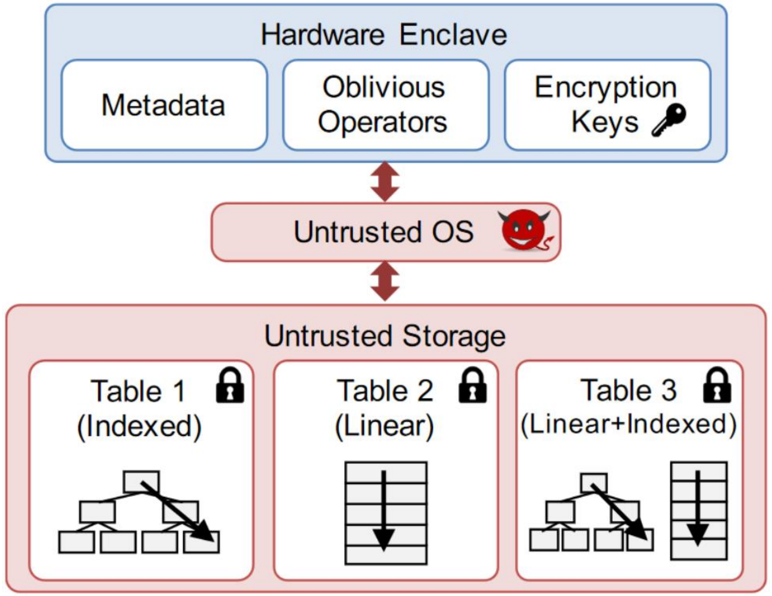
- ObliDB, a previous paper, shows a way to reduce the use of ORAM:



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Non-ORAM approaches

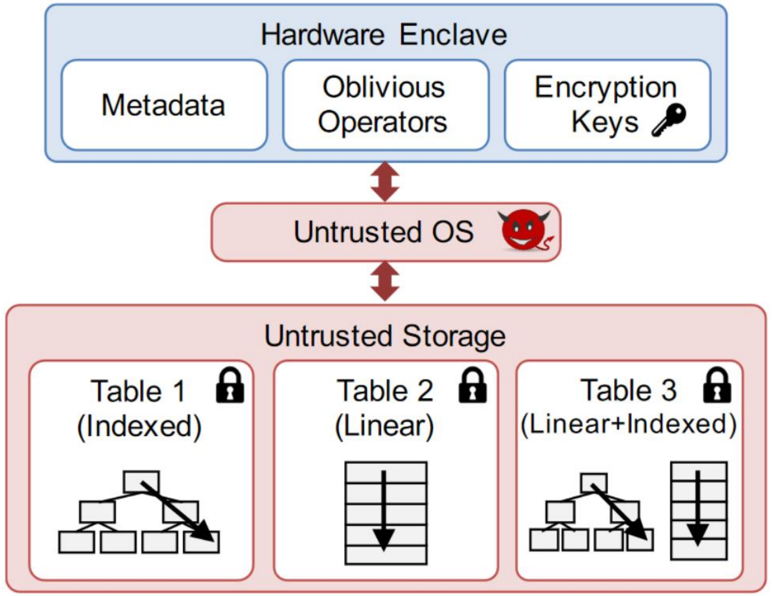
- ObliDB, a previous paper, shows a way to reduce the use of ORAM:
- Indexed tables similarly use an ORAM with a B+-tree.



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Non-ORAM approaches

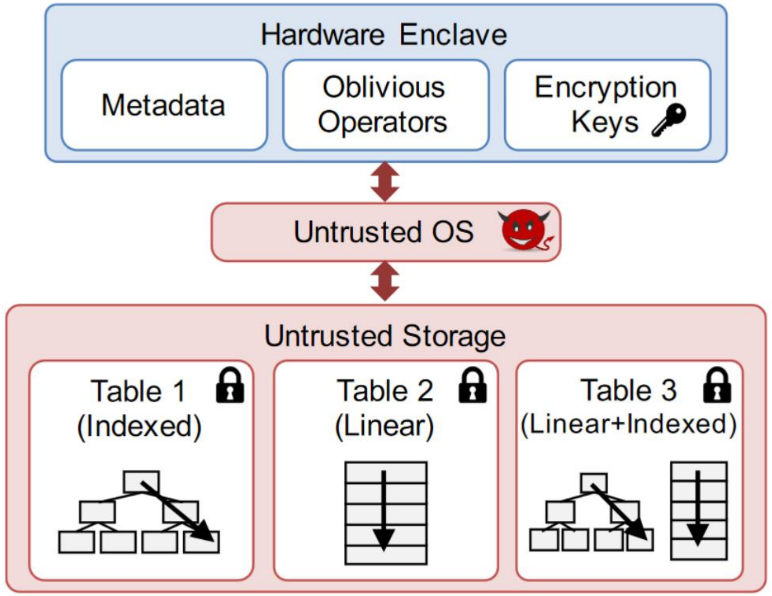
- ObliDB, a previous paper, shows a way to reduce the use of ORAM:
- Indexed tables similarly use an ORAM with a B+-tree.
- Linear tables are stored linearly – and always accessed sequentially.



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Non-ORAM approaches

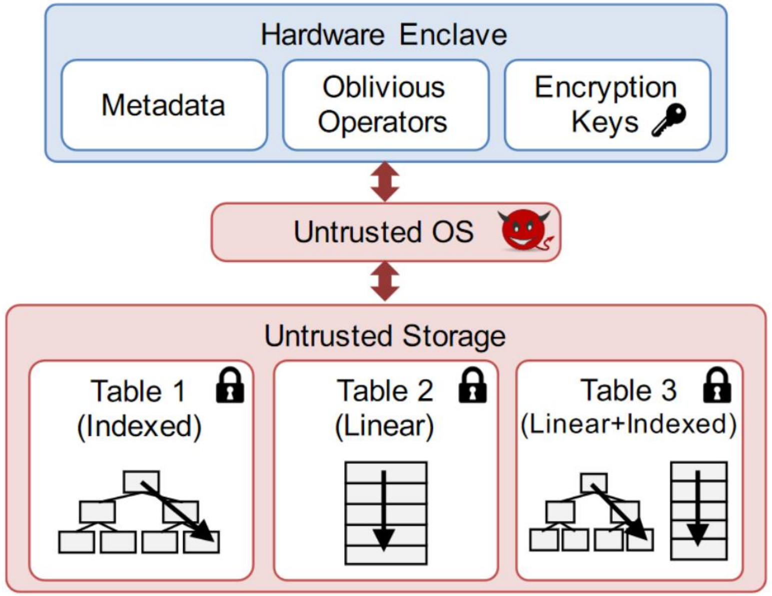
- ObliDB, a previous paper, shows a way to reduce the use of ORAM:
- Indexed tables similarly use an ORAM with a B+-tree.
- Linear tables are stored linearly – and always accessed sequentially.
- Sort-Merge join can be implemented by appending the tables to each other and using bitonic sort ($O(n \log^2 n)$ deterministic reorders)



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Non-ORAM approaches

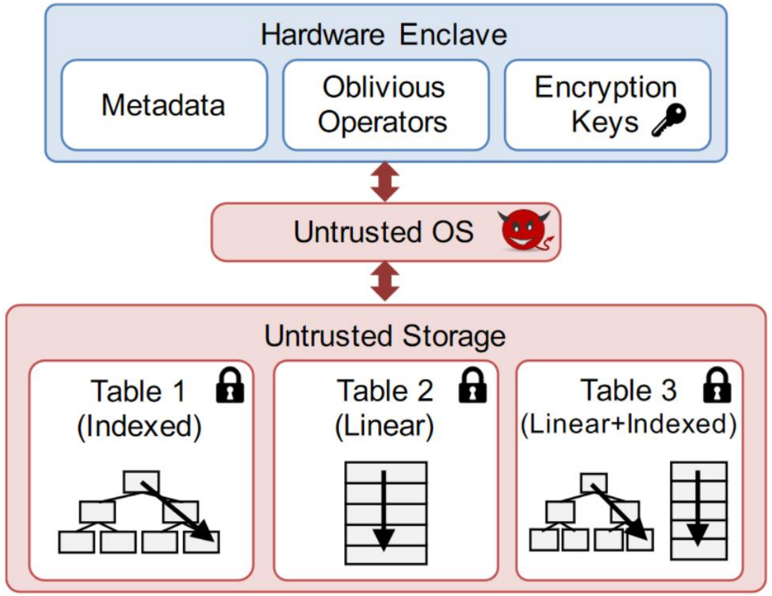
- ObliDB, a previous paper, shows a way to reduce the use of ORAM:
- Indexed tables similarly use an ORAM with a B+-tree.
- Linear tables are stored linearly – and always accessed sequentially.
- Sort-Merge join can be implemented by appending the tables to each other and using bitonic sort ($O(n \log^2 n)$ deterministic reorders)
- Index nested loop's loop can use the flat data.



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Non-ORAM approaches

- ObliDB, a previous paper, shows a way to reduce the use of ORAM:
- Indexed tables similarly use an ORAM with a B+-tree.
- Linear tables are stored linearly – and always accessed sequentially.
- Sort-Merge join can be implemented by appending the tables to each other and using bitonic sort ($O(n \log^2 n)$ deterministic reorders)
- Index nested loop's loop can use the flat data.
- Cool, but linear tables don't really help in multiway joins…



Method	Flat	Index	Both
Space	N	$\sim 4N$	$\sim 5N$
Point Read	$O(N)$	$O(\log^2 N)$	$O(\log^2 N)$
Large Read	$O(N)$	$O(N)$	$O(N)$
Insert	$O(1)$	$O(\log^2 N)$	$O(\log^2 N)$
Update	$O(N)$	$O(\log^2 N)$	$O(N)$
Delete	$O(N)$	$O(\log^2 N)$	$O(N)$

Experimental results

- Client: 18GB ram, 8-core i7
- Server: 256GB ram, 2TB disk, 8-core Xeon E5
- Communication: 1Gb bandwidth

Experimental results

- Client: 18GB ram, 8-core i7
- Server: 256GB ram, 2TB disk, 8-core Xeon E5
- Communication: 1Gb bandwidth
- ORAM storage backend: MongoDB server running insertion, deletion and updates, but no other computation or optimizations.
  - 4kb blocks

Experimental results

- Client: 18GB ram, 8-core i7
- Server: 256GB ram, 2TB disk, 8-core Xeon E5
- Communication: 1Gb bandwidth
- ORAM storage backend: MongoDB server running insertion, deletion and updates, but no other computation or optimizations.
    - 4kb blocks
- All implementations are running in non-padded mode:

Experimental results

- Client: 18GB ram, 8-core i7
- Server: 256GB ram, 2TB disk, 8-core Xeon E5
- Communication: 1Gb bandwidth
- ORAM storage backend: MongoDB server running insertion, deletion and updates, but no other computation or optimizations.
  - 4kb blocks
- All implementations are running in non-padded mode:
  - The best padding is the Cartesian Product padding which doesn't reveal the join size at all.

Experimental results

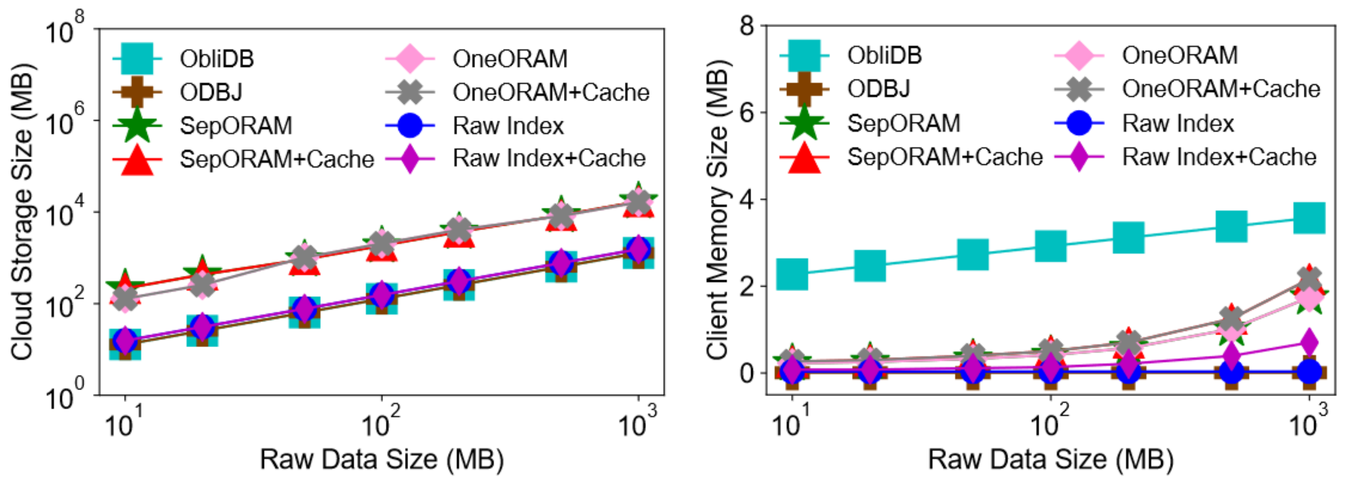
- Client: 18GB ram, 8-core i7
- Server: 256GB ram, 2TB disk, 8-core Xeon E5
- Communication: 1Gb bandwidth
- ORAM storage backend: MongoDB server running insertion, deletion and updates, but no other computation or optimizations.
  - 4kb blocks
- All implementations are running in non-padded mode:
  - The best padding is the Cartesian Product padding which doesn’t reveal the join size at all.
  - Another option is the closest power of 2 which means up to 2x overhead.

Experimental results

- The compared implementations are:
  - Separate ORAM and One ORAM
    - SMJ (Sort Merge Join)
    - INLJ (Index Nested Loop Join)
    - INLJ+Cache (with Index Caching): the client caches all index blocks above the leaf level, and due to the large fan-out in the B-tree this is a small amount of storage.
  - Baseline: Raw SMJ, INLJ, and INLJ+Cache
    - Don’t use any encryption or ORAM protocol but the data is in a remote server.
  - Previous work:
    - ObliDB, ODBJ

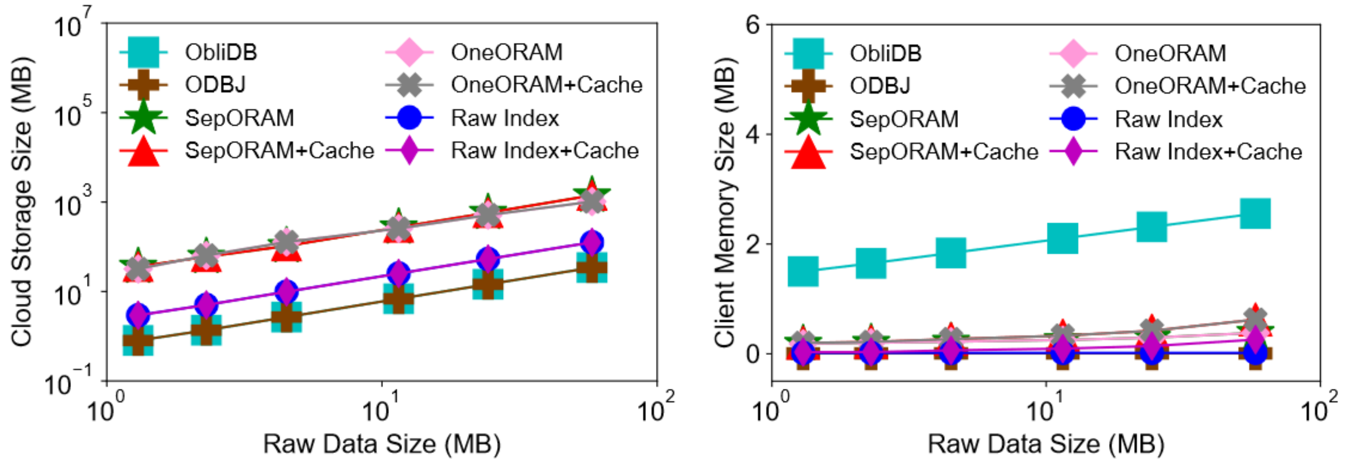
Storage and Memory

- The client memory requirement is small, and not very large even with the cache.
- The cloud storage size has noticeable (roughly 10X) overhead over a raw index, due to the use of Path-ORAM.



(a) Cloud storage size. (b) Client memory size.

Figure 7: Storage cost against raw data size on TPC-H.

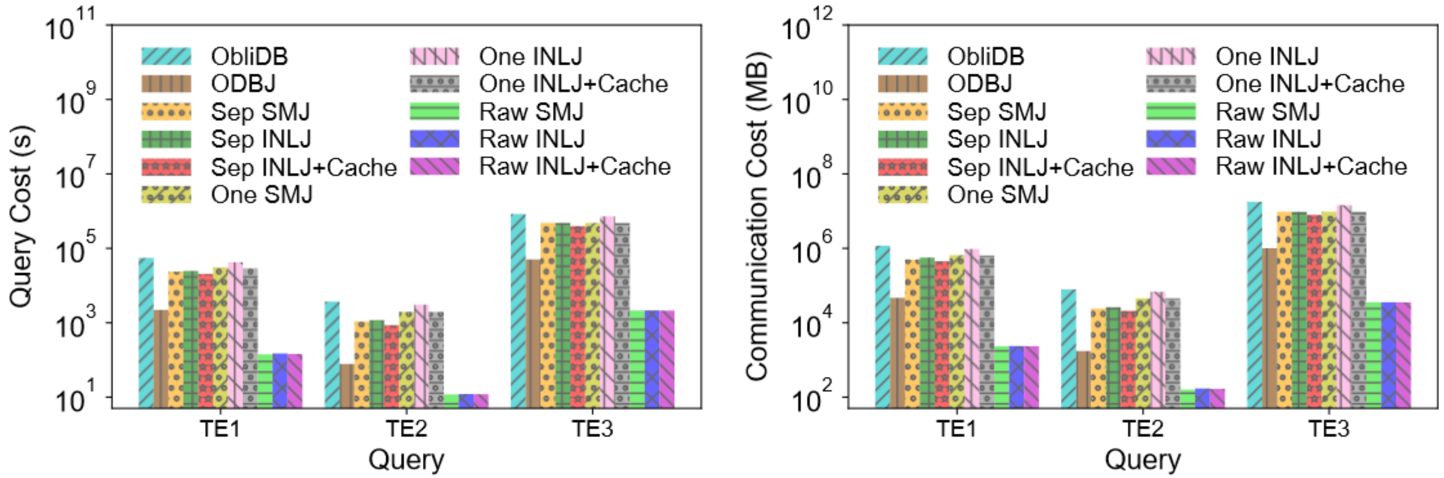


(a) Cloud storage size. (b) Client memory size.

Figure 8: Storage cost against raw data size on social graph.

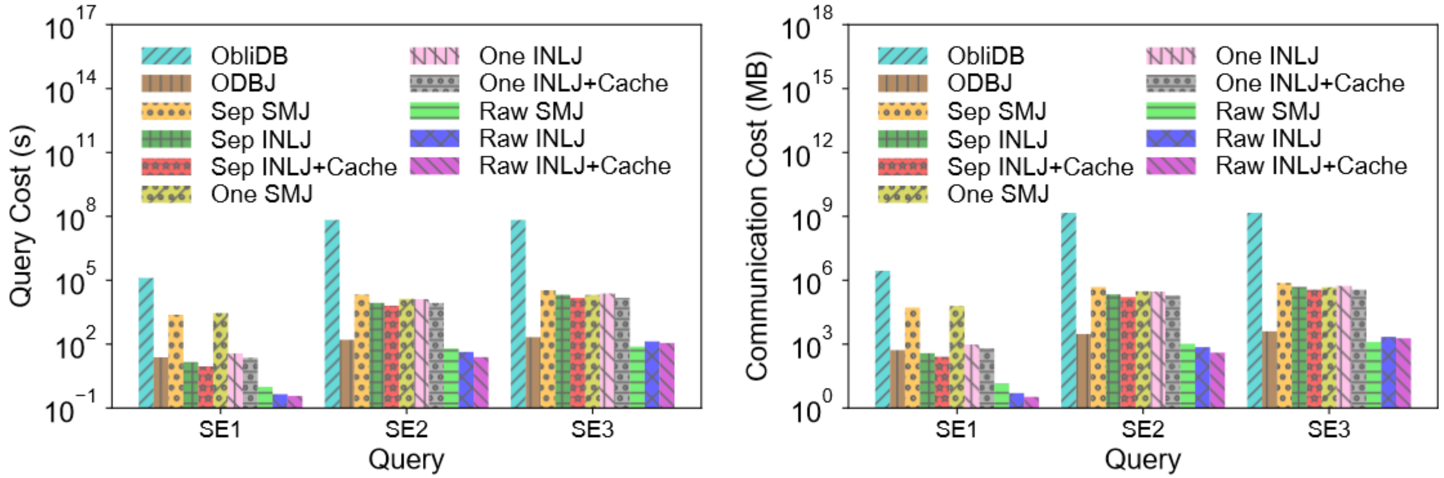
Binary Equi-Join: Performance

- Sep-INLJ is 1.2-2.6X faster than One-INLJ
- SepORAM(+Cache) is 90-450X more expensive than Raw Index(+Cache) except for SE1.
- However, the data tuples are only 100-200 bytes in TPC-H and 2 integers in social graph, and the block size of the ORAM is 4KB.



(a) Query cost. (b) Communication cost.

Figure 9: Performance of binary equi-join on TPC-H.

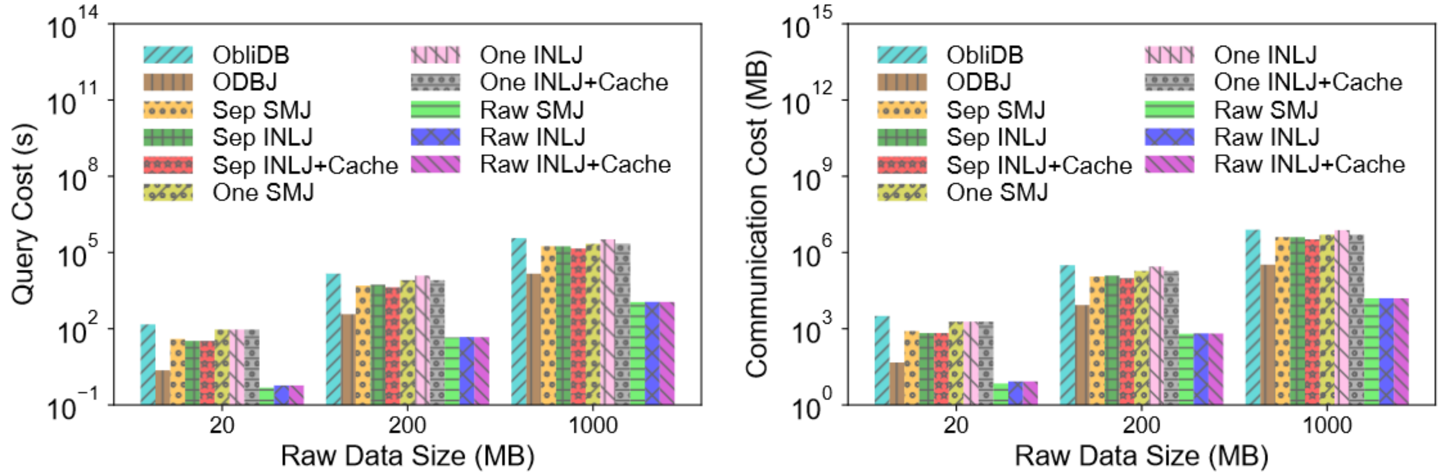


(a) Query cost. (b) Communication cost.

Figure 10: Performance of binary equi-join on social graph.

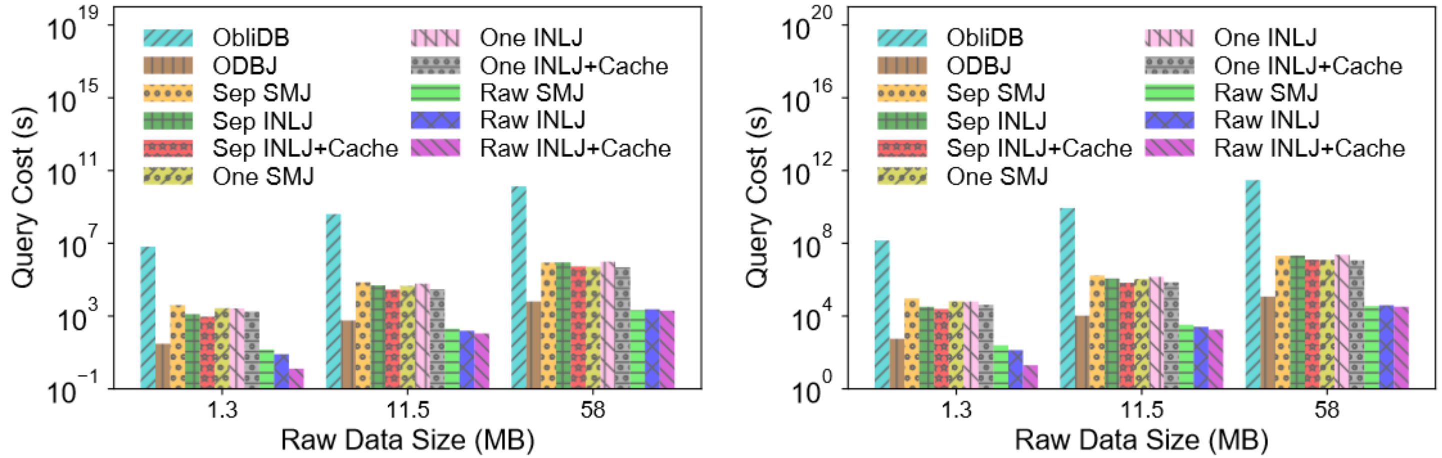
Binary Equi-Join: Scalability

- The performance difference is again because of the block size.
- The merge algorithm is scalable.



(a) Query cost. (b) Communication cost.

Figure 11: Performance of Query TE2 against raw data size.



(a) Query cost. (b) Communication cost.

Figure 12: Performance of Query SE2 against raw data size.

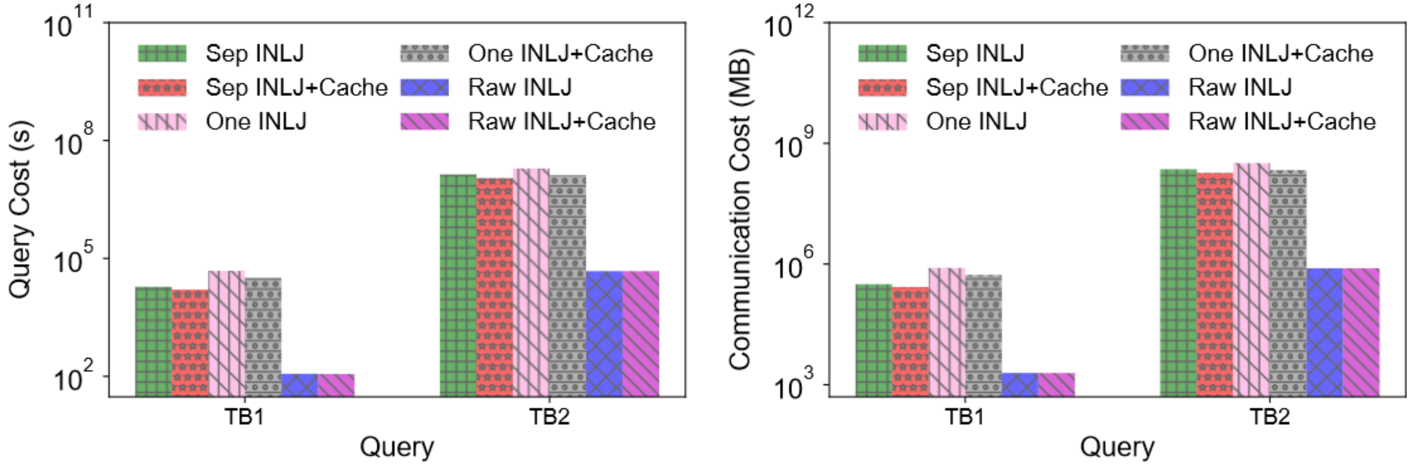
Binary Band Join: Performance

TB1 (10MB)

- Sep is 164-238X slower than Raw.
- Sep is 1.4-2.5X faster than One.
- Caching is 1.2-1.5X faster.

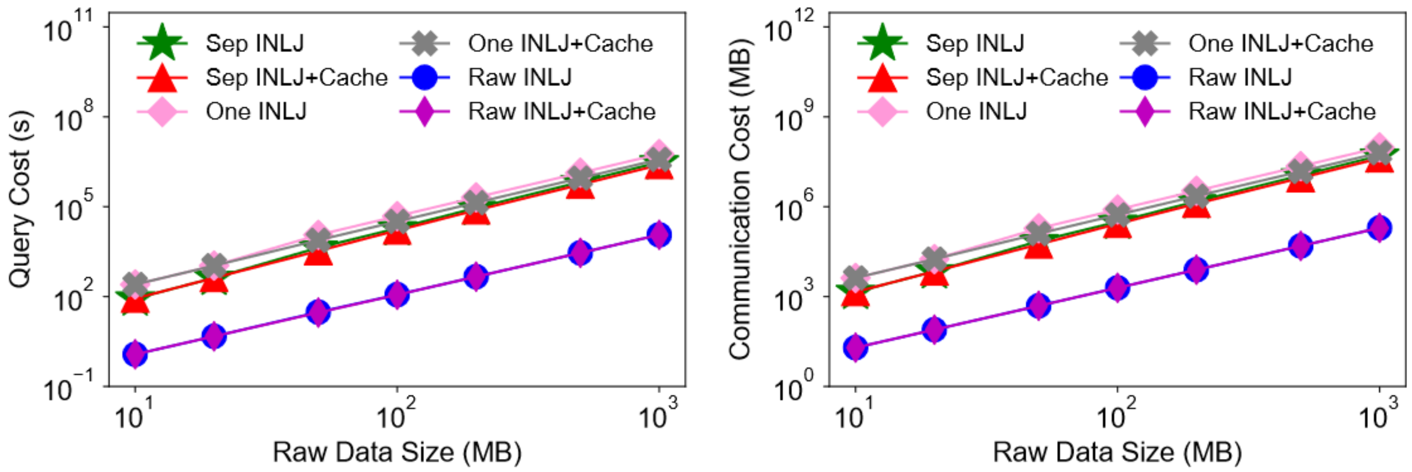
TB2 (1GB):

- Sep is 73-264X slower than Raw.
- Sep is 1.9-2.9X faster than One.
- Caching is again 1.2-1.5X faster.



(a) Query cost. (b) Communication cost.

Figure 13: Performance of band join on TPC-H.



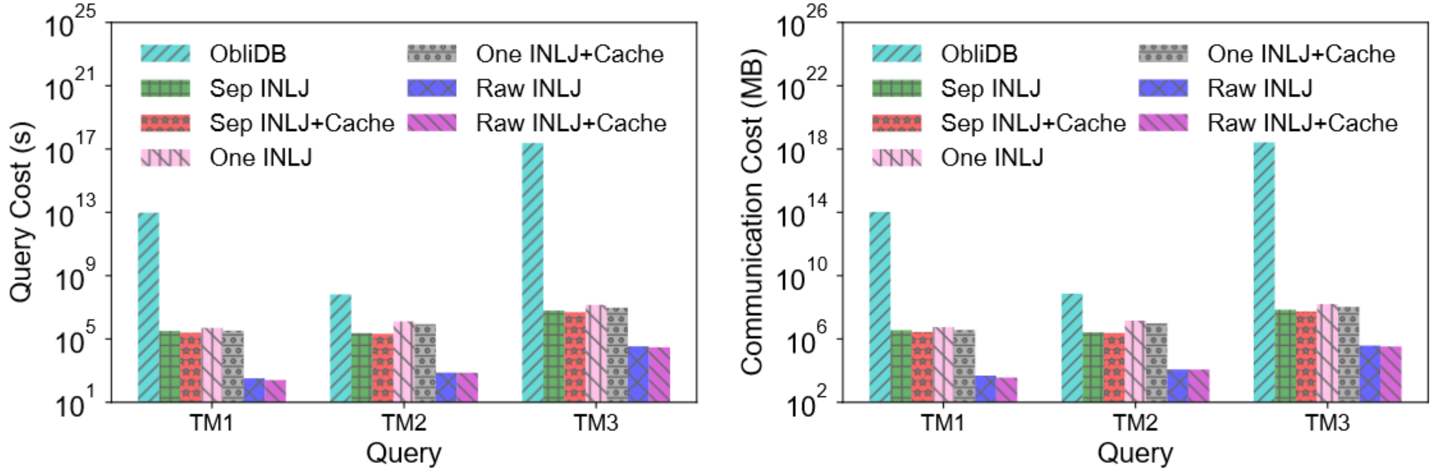
(a) Query cost. (b) Communication cost.

Figure 14: Performance of Query TB1 against raw data size.

Multiway Equi-Join: Performance

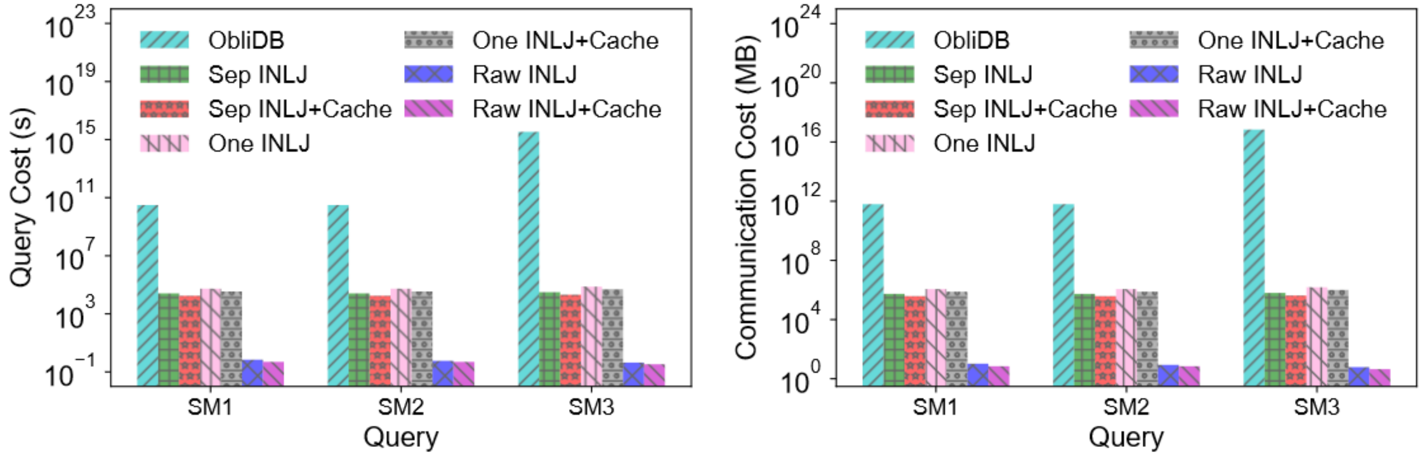
- ObliDB performs a cartesian product.
- Sep is 1.6-2.4X faster than One.
- Caching is 1.1-1.5X faster.
- Sep is <u>185-985X</u> slower than Raw in TPC-H, <u>37000-70000X</u> on social graph.
- That is because even if the join result is very small, we must pad the number of join steps to the upper bound,

$$|T_1| + 2\sum_{j=2}^{\ell} |T_j| + |R_{\mathrm{real}}|$$



(a) Query cost. (b) Communication cost.

Figure 15: Performance of multiway equi-join on TPC-H.

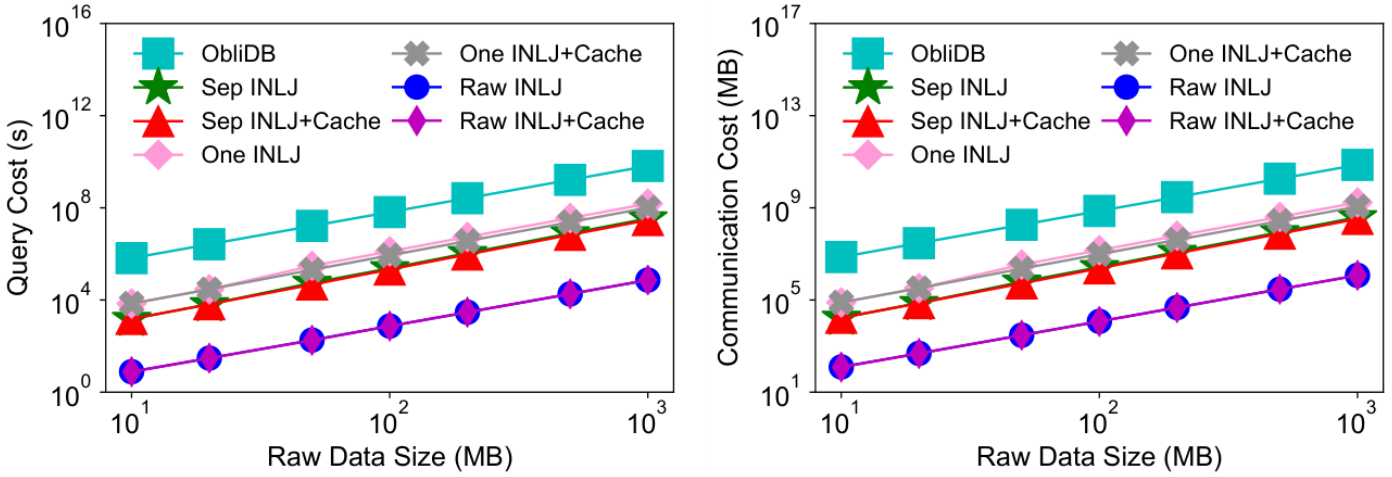


(a) Query cost. (b) Communication cost.

Figure 16: Performance of multiway equi-join on social graph.

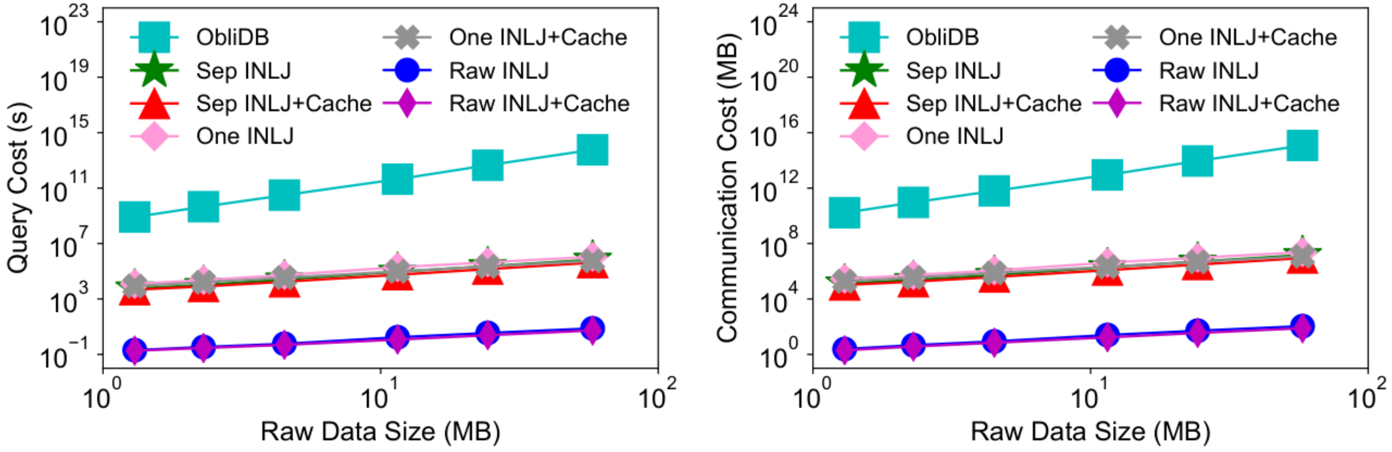
Multiway Equi-Join: Scalability

- For TM2, Sep-INLJ(+Cache) is 190-430X faster than ObliDB, and that is stable because the join result is proportional to the cartesian product.
- Sep is <u>149-469X</u> slower than raw.
- For SM2 (social graph), which has a small join result, the gap from ObliDB is $10^5 - 10^8$X.
- Sep is <u>28000-91000X</u> slower than raw.



(a) Query cost. (b) Communication cost.

Figure 17: Performance of Query TM2 against raw data size.

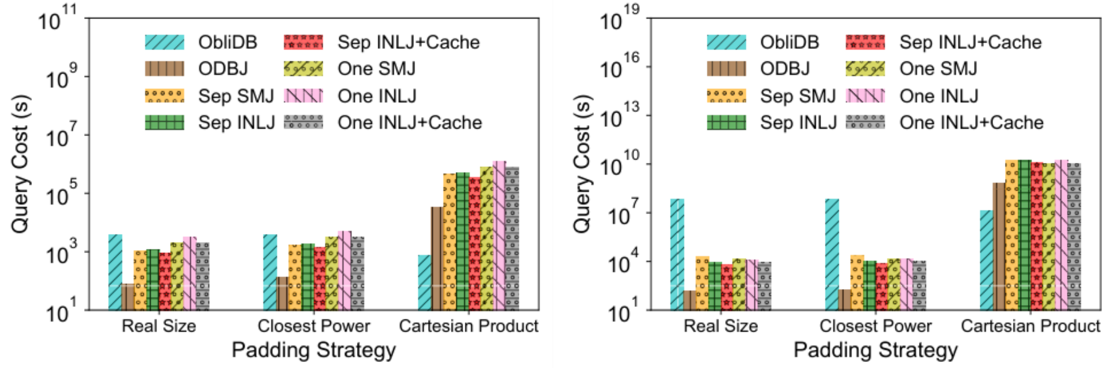


(a) Query cost. (b) Communication cost.

Figure 18: Performance of Query SM2 against raw data size.

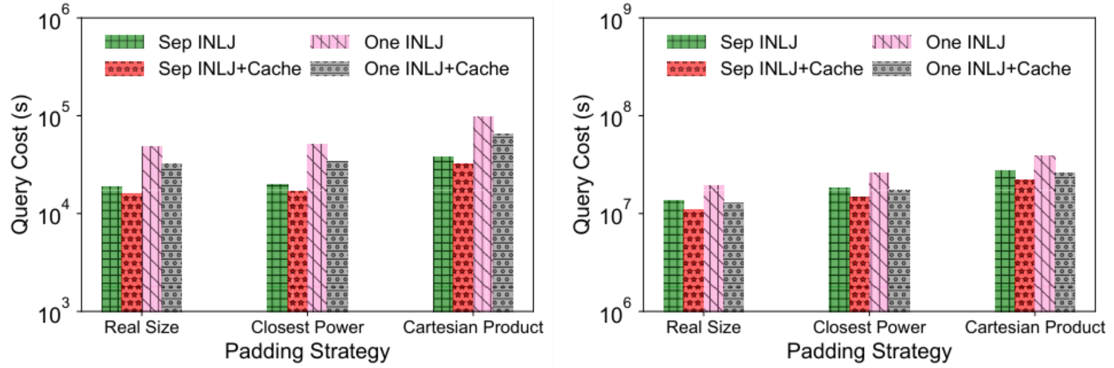
Padding Performance

- Closest power is (obviously) within 2X of Real Size.
- ObliDB is faster with Cartesian Product than it is with Real Size or Closest Product because otherwise it uses oblivious filtering.
- ObliDB is best at Cartesian Product because we have less trusted memory and use B-tree searches over ORAMs (instead of a linear database).



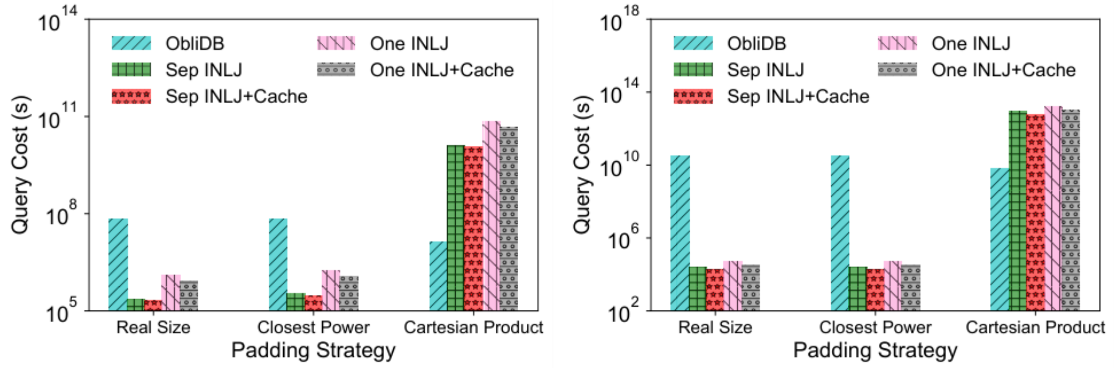
(a) Query TE2. (b) Query SE2.

Figure 19: Padded vs. non-padded mode (binary equi-join).



(a) Query TB1. (b) Query TB2.

Figure 20: Padded vs. non-padded mode (band join).



(a) Query TM2. (b) Query SM2.

Figure 21: Padded vs. non-padded mode (multiway equi-join).

Conclusions

Conclusions

- We have explored some of the most foundational join algorithms in a new light.

Conclusions

- We have explored some of the most foundational join algorithms in a new light.
- Oblivious RAM is cool and it's a thing! In fact, the Signal messenger uses it for contact discovery, along with a secure enclave.

Conclusions

- We have explored some of the most foundational join algorithms in a new light.
- Oblivious RAM is cool and it's a thing! In fact, the Signal messenger uses it for contact discovery, along with a secure enclave.
- We can learn a lot from memory access patterns. As an example, that kind of information could be leaked by a cache attack.

Conclusions

- We have explored some of the most foundational join algorithms in a new light.
- Oblivious RAM is cool and it's a thing! In fact, the Signal messenger uses it for contact discovery, along with a secure enclave.
- We can learn a lot from memory access patterns. As an example, that kind of information could be leaked by a cache attack.
- Oblivious RAM is a nice and useful abstraction but when using it we are still responsible to make the number of accesses deterministic.

Conclusions

- We have explored some of the most foundational join algorithms in a new light.
- Oblivious RAM is cool and it’s a thing! In fact, the Signal messenger uses it for contact discovery, along with a secure enclave.
- We can learn a lot from memory access patterns. As an example, that kind of information could be leaked by a cache attack.
- Oblivious RAM is a nice and useful abstraction but when using it we are still responsible to make the number of accesses deterministic.
- Even by timing how long the server takes to respond, we could measure how many join results there are, which is an interesting attack vector.