SODA: A Set of Fast Oblivious Algorithms in Distributed Secure Data Analytics

Xiang Li
Tsinghua University
lixiang20@mails.tsinghua.edu.cn

Nuozhou Sun
Tsinghua University
snz21@mails.tsinghua.edu.cn

Yunqian Luo
Tsinghua University
luoyq19@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
gaomy@tsinghua.edu.cn

1 OBLIVIOUS FILTER

Algorithm 1: Oblivious Global Filter

$\mathcal{B}_{all} \leftarrow []$;
for $i \leftarrow 0$ **to** $N_{in} - 1$ **do**
  $\mathcal{B}_{all}[i] \leftarrow$ `OFilterStage1`$(\mathcal{P}_{in}[i], N_{in})$
for $i \leftarrow 0$ **to** $N_{out} - 1$ **do**
  $\mathcal{P}_{out}[j] \leftarrow$ `OFilterStage2`$(\mathcal{B}_{all}[..][j])$

Algorithm 2: Oblivious Global Filter: Stage 1

Input: Local data partition $\mathcal{P}$, total number of partitions N
Output: N buckets $\mathcal{B}$.

foreach $d \in \mathcal{P}$ **do** // randomly assign records to buckets.
  d.bucket $\leftarrow$ `GetRandom()` mod N;
$\mathcal{B} \leftarrow$ `BuildBuckets`$(\mathcal{P}, N)$;
return $\mathcal{B}$;
function `BuildBuckets`$(\mathcal{P}, l, N)$**:**
  `OSort`$(\mathcal{P}$, (_.bucket, _.valid, _.key));
  /* Assign locations in buckets. */
  $b \leftarrow \perp$; $p \leftarrow \perp$;
  foreach $d \in \mathcal{P}$ **do**
    $c \leftarrow d$.bucket $\neq b$; // encounter a new bucket?
    $b \leftarrow d$.bucket;
    `cmov`$(c, p, l \times b)$; // starting location of a new bucket.
    d.pos $\leftarrow p$; $p \leftarrow p + 1$;
  /* Place records to assigned locations. */
  Pad $\mathcal{P}$ to length $l \cdot N$;
  `ODistribute`$(\mathcal{P}$, _.pos$)$; // invalid records at end of each bucket.
  $\mathcal{B} \leftarrow$ Chunk $\mathcal{P}$ into buckets of size l;
  return $\mathcal{B}$;

Algorithm 3: Oblivious Global Filter: Stage 2

Input: Buckets $\mathcal{B}$ collected from all nodes
Output: Partition $\mathcal{P}$ to be processed later by this node

/* Note that each bucket is already sorted by k */
$\mathcal{D} \leftarrow$ `OMerge`$(\mathcal{B}, (\neg is\_valid(\_.v), \_.k))$; // merge by k
$\mathcal{P} \leftarrow$ Directly remove all dummy values at the end of $\mathcal{D}$;
return $\mathcal{P}$;

Algorithm 4: (Pseudo-)Random Communication Buckets

function `BuildBuckets`$(\mathcal{P}, N)$**:**
  Input: Data partition $\mathcal{P}$ with (pseudo-)randomly assigned bucket field per record, total number of partitions N.
  Output: N buckets $\mathcal{B}$.
  /* In implementation, we fix the error rate $Ne^{-k^2/3}$ to compute the pad length */
  Compute the padding size $l \leftarrow \frac{|\mathcal{P}|}{N} + k\sqrt{\frac{|\mathcal{P}|}{N}}$;
  `OSort`$(\mathcal{P}$, (_.bucket, _.valid, _.key));
  /* Assign locations in buckets. */
  $b \leftarrow \perp$; $p \leftarrow \perp$;
  foreach $d \in \mathcal{P}$ **do**
    $c \leftarrow d$.bucket $\neq b$; // encounter a new bucket?
    $b \leftarrow d$.bucket;
    `cmov`$(c, p, l \times b)$; // starting location of a new bucket.
    d.pos $\leftarrow p$; $p \leftarrow p + 1$;
  /* Place records to assigned locations. */
  Pad $\mathcal{P}$ to length $l \cdot N$;
  `ODistribute`$(\mathcal{P}$, _.pos$)$; // invalid records at end of each bucket.
  $\mathcal{B} \leftarrow$ Chunk $\mathcal{P}$ into buckets of size l;
  return $\mathcal{B}$;

2 OBLIVIOUS AGGREGATE

Algorithm 5: Oblivious Aggregate

/* In parallel, each node i processes its own partition. */
$\mathcal{B}_{all} \leftarrow []$;
for $i \leftarrow 0$ **to** $N_{in} - 1$ **do**
  $\mathcal{B}_{all}[i] \leftarrow$ `OAggStage1`$(\mathcal{F}, \mathcal{P}_{in}[i], i, false)$

/* In parallel, each node j collects its partition through network and do further processing. */
for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**
  $\mathcal{P}_{out}[j] \leftarrow$ `OAggStage2`$(\mathcal{F}, \mathcal{B}_{all}[..][j], true)$

Algorithm 6: Oblivious Aggregate: Stage 1

function `OAggStage1`$(\mathcal{F}, \mathcal{P}, i, N_{out}, should\_assign\_loc)$**:**
  Input: Local data partition $\mathcal{P}$, total number of partitions N. Aggregate function $\mathcal{F}$
  Output: N buckets $\mathcal{B}$.
  `OSort`$(\mathcal{P}, \_.\text{key})$; // sort by key.
  if $should\_assign\_loc$ **then**
    /* $\mathcal{P}$ has two more fields: location l, i.e., the current item index, and original partition id i. */
    $\mathcal{P} \leftarrow$ `AssignLoc`$(\mathcal{P}, i)$;
  else
    $\mathcal{P} \leftarrow \mathcal{P}$;
  /* Perform aggregate and mark dummy items. */
  $\mathcal{P} \leftarrow$ `Aggregate`$(\mathcal{F}, \mathcal{P})$;
  /* Assign buckets. */
  nonce $\leftarrow 0$;
  foreach $d \in \mathcal{P}$ **do**
    $h \leftarrow$ `Hash`$(d.\text{key})$; // use real key.
    `cmov`$(\neg d.\text{valid}, h,$ `Hash`$(d.\text{key} \| \text{nonce}))$; // use dummy key.
    $d.\text{bucket} \leftarrow h \bmod N$;
    nonce $\leftarrow$ nonce $+ 1$;
  $\mathcal{B} \leftarrow$ `BuildBuckets`$(\mathcal{P}, N)$;
  return $\mathcal{B}$;

function `Aggregate`$(\mathcal{F}, \mathcal{D})$**:**
  Input: Aggregate function $\mathcal{F}$, data $\mathcal{D}$
  Output: Data after aggregate $\mathcal{D}$
  /* Do further aggregate */
  $acc \leftarrow \mathcal{D}[0].v$;
  for $i \leftarrow 1$ **to** $|\mathcal{D}| - 1$ **do**
    $c \leftarrow acc \neq \perp \wedge \mathcal{D}[i].v \neq \perp$;
    `cmov`$(acc = \perp, acc, \mathcal{D}[i].v)$;
    `cmov`$(c, acc, \mathcal{F}(acc, \mathcal{D}[i].v))$;
    $c \leftarrow \mathcal{D}[i-1].k \neq \mathcal{D}[i].k$;
    `cmov`$(c, acc, \mathcal{D}[i].v)$;
    `cmov`$(\neg c, \mathcal{D}[i-1].v, \perp)$;
    $\mathcal{D}[i].v \leftarrow acc$;
  return $\mathcal{D}$;

Algorithm 7: Oblivious Aggregate: Stage 2

function `OAggStage2`$(\mathcal{F}, \mathcal{B}, should\_remove\_dummy)$**:**
  Input: Aggregate function $\mathcal{F}$, buckets $\mathcal{B}$ collected from all nodes
  Output: Partition $\mathcal{P}$ to be processed later by this node
  /* Note that each bucket is already sorted by k */
  $\mathcal{D} \leftarrow$ `OMerge`$(\mathcal{B}, \_.k, is\_valid(\_.v))$; // merge by k, and invalid values are at the front of the group.
  /* Do further aggregate */
  $\mathcal{D} \leftarrow$ `Aggregate`$(\mathcal{F}, \mathcal{D})$;
  if $should\_remove\_dummy$ **then**
    /* Remove all dummy values */
    $\mathcal{P} \leftarrow$ `OLocalFilter`$(\mathcal{D})$;
  else
    $\mathcal{P} \leftarrow \mathcal{D}$;
  return $\mathcal{P}$;

3 OBLIVIOUS JOIN

Algorithm 8: Oblivious Join

Oblivious global filter $\mathcal{P}_{A-in}$ and $\mathcal{P}_{B-in}$; // shuffle.

/* Perform oblivious aggregate, and the aggregate function is count. */

$\mathcal{B}_{A-agg}, \mathcal{B}_{B-agg} \leftarrow [], []$;

for $i \leftarrow 0$ **to** $N_{in} - 1$ **do**

  $\mathcal{B}_{A-agg}[i], \pi_A[i] \leftarrow$ `OAggStage1`$(count, \mathcal{P}_{A-in}[i], i, true)$;

  $\mathcal{B}_{B-agg}[i], \pi_B[i] \leftarrow$ `OAggStage1`$(count, \mathcal{P}_{B-in}[i], i, true)$;

  /* Note that $\mathcal{P}_{A-in}$ [i] and $\mathcal{P}_{B-in}$ [i] is sorted by k at this point. */

$\beta_A, \beta_B \leftarrow 0, 0$;

for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

  $\mathcal{P}_{A-agg}[j] \leftarrow$ `OAggStage2`$(count, \mathcal{B}_{A-agg}[..][j], false)$;

  $\mathcal{P}_{B-agg}[j] \leftarrow$ `OAggStage2`$(count, \mathcal{B}_{B-agg}[..][j], false)$;

  /* $\mathcal{P}_{A-agg}$ and $\mathcal{P}_{B-agg}$ has fields k, v, i, l */

  $a_A, cnt_A \leftarrow$ `OJoinStage0`$(\mathcal{P}_{A-agg}[j])$;

  $a_B, cnt_B \leftarrow$ `OJoinStage0`$(\mathcal{P}_{B-agg}[j])$;

  /* Need each node to broadcast their a_A, a_B, cnt_A and cnt_B. */

  $\alpha_A, \alpha_B \leftarrow \max(\alpha_A, a_A), \max(\alpha_B, a_B)$;

  $\beta_A, \beta_B \leftarrow \beta_A + cnt_A, \beta_B + cnt_B$;

$\mathcal{I} \leftarrow []$;

for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

  $bin\_num, bin\_size, N_{out}', acc_{prod}, \mathcal{P}_{agg}[j] \leftarrow$ `OJoinStage1`$(\mathcal{P}_{A-agg}[j], \mathcal{P}_{B-agg}[j], \alpha_A, \alpha_B, \beta_A, \beta_B, N_{out})$;

  /* $\mathcal{P}_{agg}$ has fields: k, v, i, l, t, b. */

  Append $(bin\_num, bin\_size, acc_{prod})$ to $\mathcal{I}$;

$\mathcal{B}_{A-agg}, \mathcal{B}_{B-agg} \leftarrow [], []$;

for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

  /* Coordinate the last bin number and update all bin numbers */

  $\mathcal{P}_{agg}[j], \mathcal{D}[j], \mathcal{S}[j], \mathcal{L}[j], \mathcal{S}_{rem}[j] \leftarrow$ `OJoinStage2`$(\mathcal{P}_{agg}[j], [\alpha_A, \alpha_B], j, \mathcal{I}, N_{out}')$;

for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

  /* Aggregate column sum and adjust partition number by column. */

  $\mathcal{L}_{prev} \leftarrow \perp$;

  if $j = 0$ **then**

    $\mathcal{S}_g, \mathcal{S}_{rem} \leftarrow$ `AssignRemBin`$(\mathcal{S}_{rem}, N_{out}')$; // collect $\mathcal{S}_{rem}$ from other nodes and broadcast the new one.

  $\mathcal{S}_g, \mathcal{S}[j], \mathcal{L}[j] \leftarrow$ `OJoinStage3`$(\mathcal{S}_g, \mathcal{S}[j], \mathcal{L}_{prev}, \mathcal{L}[j], \mathcal{S}_{rem}[j])$;

  $\mathcal{L}_{prev} \leftarrow \mathcal{L}[j]$; // Send to node $j+1$.

  Send $\mathcal{S}_g$ to node $j+1$;

$\gamma \leftarrow \sum_i \mathcal{S}_g[i].1$; // computed by node $N_{out} - 1$ and broadcast.

for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

  $\mathcal{B}_{A-agg}[j], \mathcal{B}_{B-agg}[j] \leftarrow$ Split the buckets returned by `OJoinStage4`$(\mathcal{P}_{agg}[j], \mathcal{D}[j], \mathcal{S}[j], \mathcal{L}[j], \mathcal{S}_{rem}[j])$;

/* Assign the destination partition number to items in input partitions, pad according to the public information, and build buckets. */

$\mathcal{B}_{all} \leftarrow []$;

for $i \leftarrow 0$ **to** $N_{in} - 1$ **do**

  $\mathcal{B}_{all}[i] \leftarrow$ `OJoinStage5`$([\mathcal{P}_{A-in}[i], \mathcal{P}_{B-in}[i]], [\mathcal{B}_{A-agg}[..][i], \mathcal{B}_{B-agg}[..][i]], [\beta_A, \beta_B], N_{in}, N_{out}')$

/* Do join. */

for $j \leftarrow 0$ **to** $N_{out}' - 1$ **do**

  $\mathcal{P}_{out}[j] \leftarrow$ `OJoinStage6`$(\mathcal{B}_{all}[..][j], \gamma / N_{out}' + \alpha_A \cdot \alpha_B)$;

Algorithm 9: Oblivious Join: Stage 0

function `OJoinStage0`$(\mathcal{P})$:

  Input: Partition after aggregate $\mathcal{P}$

  Output: The max count of items belonging to a key a, the total count of items in this partition cnt

  $a \leftarrow 0$;

  $cnt \leftarrow 0$;

  for $i \leftarrow 0$ **to** $|Part| - 1$ **do**

    /* After `OAggStage2`, only one value correspond to the same key is valid. */

    $c \leftarrow \mathcal{P}[i].v \neq \perp$;

    `cmov`$(c, a, \max(a, \mathcal{P}[i].v))$;

    `cmov`$(c, cnt, cnt + \mathcal{P}[i].v)$;

  return a, cnt;

Algorithm 10: Oblivious Join: Stage 1

function $\texttt{OJoinStage1}(\mathcal{P}_A, \mathcal{P}_B, \alpha_A, \alpha_B, \beta_A, \beta_B, N_{out})$:

  Input: Partitions of two tables $\mathcal{P}_A$ and $\mathcal{P}_B$, statistical information $\alpha_A, \alpha_B, \beta_A, \beta_B$, expected # of nodes after join N_{out}

  Output: # of nodes after join N_{out}', union partition $\mathcal{P}$

  Add t field denoting table id to all items in $\mathcal{P}_A$ and $\mathcal{P}_B$;

  // distinguish the items with the same key in two tables.

  Alias v field as $v.0/v.1$ for A/B, and pad $v.1/v.0$ respectively;

  /* Items in table B are behind those in table A with the same keys. */

  $\mathcal{P} \leftarrow \texttt{OMerge}([\mathcal{P}_A, \mathcal{P}_B], (\_.k, \_.t, is\_valid(\_.v)))$; // $\mathcal{P}_A$ and $\mathcal{P}_B$ are already sorted by k.

  if $(\alpha_A + \alpha_B)/(\beta_A + \beta_B) > 1/N_{out}$ **then**

    $N_{out}' \leftarrow \lceil(\beta_A + \beta_B)/(\alpha_A + \alpha_B)\rceil$;

  else

    $N_{out}' \leftarrow N_{out}$;

  /* Invalid items which do not have counterparts to join. */

  for $i \leftarrow 0$ **to** $|\mathcal{P}| - 1$ **do**

    $c \leftarrow \mathcal{P}[i-1].k \neq \mathcal{P}[i].k$; // for out-of-bound i it is true.

    $\texttt{cmov}(c \land acc.0 * acc.1 > 0, \mathcal{P}[i-1].v, acc)$;

    $\texttt{cmov}(c, acc, (\bot, \bot))$;

    $acc \leftarrow acc + \mathcal{P}[i].v$; // $\bot$ can be seen as 0.

    $\mathcal{P}[i].v \leftarrow (\bot, \bot)$;

  $\texttt{cmov}(acc.0 * acc.1 > 0, \mathcal{P}[|\mathcal{P}| - 1].v, acc)$;

  /* First-level assignment. */

  $cap \leftarrow \alpha_A + \alpha_B$;

  $bin\_num \leftarrow 0$;

  $bin\_size \leftarrow 0$;

  $acc_{prod} \leftarrow 0$;

  for $i \leftarrow |\mathcal{P}| - 1$ **to** 0 **do**

    $c \leftarrow \mathcal{P}[i].k \neq \mathcal{P}[i+1].k$; // for out-of-bound i it is true.

    $\texttt{cmov}(\neg c, \mathcal{P}[i].v, \mathcal{P}[i+1].v)$; // fill the vacant.

    /* Avoid repeated count. */

    $c_0 \leftarrow c \land \mathcal{P}[i].v \neq (\bot, \bot)$;

    $bin\_num_{new}, bin\_size \leftarrow \texttt{NextFit}(cap, c_0, \mathcal{P}[i].v.0 + \mathcal{P}[i].v.1, bin\_num, bin\_size)$;

    $c_1 \leftarrow bin\_num_{new} > bin\_num$; // a new bin is created

    $\mathcal{P}[i].b, bin\_num \leftarrow bin\_num_{new}, bin\_num_{new}$;

    /* Compute accumulated product for normal items. */

    $\mathcal{P}[i].a \leftarrow 0$; // add new field a, equal to 0 by default.

    $\texttt{cmov}(c_0 \land c_1, \mathcal{P}[i+1].a, acc_{prod})$;

    $\texttt{cmov}(c_0 \land c_1, acc_{prod}, \mathcal{P}[i].v.0 \cdot \mathcal{P}[i].v.1)$;

    $\texttt{cmov}(c_0 \land \neg c_1, acc_{prod}, acc_{prod} + \mathcal{P}[i].v.0 \cdot \mathcal{P}[i].v.1)$;

  return $bin\_num, bin\_size, N_{out}', acc_{prod}, \mathcal{P}$;

function $\texttt{NextFit}(cap, c, w, bin\_num, bin\_size, n = bin\_num + 1)$:

  Input: Bin capacity cap, condition when bin state changes c, item weight w, bin info bin_num and bin_size, and set bin number n, equal to $bin\_num + 1$ by default

  Output: changed bin_num and bin_size

  /* Try to place in the old bin, the bin capacity is cap. */

  $tmp \leftarrow bin\_size + w$; // $\bot$ can be viewed as 0 here.

  $\texttt{cmov}(c \land tmp \leq cap, bin\_size, tmp)$;

  /* Open a new bin if space limited. */

  $\texttt{cmov}(c \land tmp > cap, bin\_num, n)$;

  $\texttt{cmov}(c \land tmp > cap, bin\_size, w)$;

  return bin_num, bin_size;

Algorithm 11: Oblivious Join: Stage 2

function $\texttt{OJoinStage2}(\mathcal{P}, \alpha, j, \mathcal{I}, N_{out})$:

  Input: $\mathcal{P}$, α, and N_{out}, same meaning as in $\texttt{OJoinStage1}$, current agg partition id j, last bin information of all partitions $\mathcal{I}$

  Output: Partition after modification $\mathcal{P}$, intermediate result for sum $\mathcal{D}$, product sum of columns $\mathcal{S}$, location of last bin l, last row of bins $\mathcal{S}_{\text{rem}}$

  $cap \leftarrow \alpha[0] + \alpha[1]$;

  $\mathcal{P} \leftarrow \texttt{CoordBinNum}(\mathcal{P}, cap, j, \mathcal{I})$;

  /* Compute accumulated product for last bins. */

  $j_s, j_{sl}, acc_{prod} \leftarrow 0, 0, 0$;

  for $i \leftarrow 0$ **to** $|\mathcal{I}| - 1$ **do**

    $\texttt{cmov}(\mathcal{I}[i].0 \neq \mathcal{I}[i-1].0, j_s, i)$; // false for out-of-bound i.

    $\texttt{cmov}(j_s = i \land j_s \leq j, acc_{prod}, \mathcal{I}[i].2)$;

    $\texttt{cmov}(j_s \neq i \land j_s \leq j, acc_{prod}, acc_{prod} + \mathcal{I}[i].2)$;

    $\texttt{cmov}(j_s \leq j, j_{sl}, j_s)$;

  $\texttt{cmov}(j_{sl} \neq j, acc_{prod}, -acc_{prod})$;

  $\mathcal{P}[0].a \leftarrow acc_{prod}$; // items with non-positive number does not involve 2nd assignment.

  /* Items with the same b share the same a. */

  for $i \leftarrow 1$ **to** $|\mathcal{P}| - 1$ **do**

    $\texttt{cmov}(\mathcal{P}[i].b = \mathcal{P}[i-1].b, \mathcal{P}[i].a, \mathcal{P}[i-1].a)$;

  /* Begin second level assignment to balance the results after join. */

  $idx \leftarrow 0$; // mark and count the representitives of bins.

  $l \leftarrow -1$; // whether the last bin involves second assignment.

  for $i \leftarrow 0$ **to** $|\mathcal{P}| - 1$ **do**

    $c \leftarrow \mathcal{P}[i].b \neq \mathcal{P}[i-1].b \land \mathcal{P}[i].a > 0$;

    $\mathcal{D}[i] \leftarrow (i, \bot, 0)$; // true for out-of-bound i.

    $\texttt{cmov}(c, \mathcal{D}[i], (i, idx, \mathcal{P}[i].a))$;

    $\texttt{cmov}(c, idx, idx + 1)$;

  /* Summarize the accumulated product by column. */

  $\mathcal{S} \leftarrow [(0,0), (1,0), (2,0), (3,0), \cdots, (N_{out} - 1, 0)]$;

  $\text{cur} \leftarrow 0$;

  $tmp \leftarrow [(0, \bot, \bot), (1, \bot, \bot), \cdots, (N_{out} - 1, \bot, \bot)]$;

  for $i \leftarrow 0$ **to** $\lceil |\mathcal{D}|/N_{out} \rceil - 1$ **do**

    $\text{chunk} \leftarrow \mathcal{D}[i \times N_{out} : (i+1) \times N_{out}]$; // reference

    foreach $d \in \text{chunk}$ **do**

      $\texttt{cmov}(d.1 \neq \bot, d.1, \text{cur mod } N_{out})$;

      $\texttt{cmov}(d.1 \neq \bot, \text{cur}, \text{cur} + 1)$;

    $\texttt{OSort}(\text{chunk}, \_.1)$;

    $\texttt{OPlace}(\text{chunk}, \_.1)$;

    for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

      $\texttt{cmov}(\text{chunk}[j].1 \neq \bot \land tmp[j].1 = \bot, tmp[j].1, \text{chunk}[j].2)$;

    $\texttt{OSort}(\mathcal{S}, \_.1)$; // assume it is in descending order.

    $\texttt{OSort}(tmp, \_.1)$; // assume it is in ascending order, sort dummy to the end.

    $c \leftarrow \text{cur} \geq N_{out}$;

    for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

      $\texttt{cmov}(c, \mathcal{S}[j].1, \mathcal{S}[j].1 + tmp[j].1)$;

      $\texttt{cmov}(c, tmp[j].2, \mathcal{S}[j].0)$;

    $\texttt{OSort}(tmp, \_.0)$; // assume it is in ascending order.

    $\text{cur} \leftarrow \text{cur mod } N_{out}$;

    for $j \leftarrow 0$ **to** $N_{out} - 1$ **do**

      $\texttt{cmov}(c \land j < \text{cur}, tmp[j].1, \text{chunk}[j].2)$;

      $\texttt{cmov}(c \land j \geq \text{cur}, tmp[j].1, \bot)$;

      $\texttt{cmov}(c, \text{chunk}[j].2, tmp[j].2)$;

      $\texttt{cmov}(\neg c, \text{chunk}[j].2, \bot)$;

      $\texttt{cmov}(c \land \mathcal{P}[0].a > 0 \land l = -1, l, \text{chunk}[0].2)$;

      // record the original column num for the last bin.

  $\texttt{cmov}(\mathcal{P}[0].a > 0 \land l = -1, l, -2)$; // the last bin is in $\mathcal{S}_{\text{rem}}$.

  $\mathcal{S}_{\text{rem}} \leftarrow tmp[0 : N_{out}]$; // strip field 2.

  return $\mathcal{P}, \mathcal{D}, \mathcal{S}, l, \mathcal{S}_{\text{rem}}$;

Algorithm 12: Assign Remainder Bins

function AssignRemBin($\mathcal{S}_{rem}$, N_{out}):

Input: Remaining bin info $\mathcal{S}_{rem}$, number of nodes after join N_{out}

Output: Initial column sum info $\mathcal{S}$, remaining bin info after assignment $\mathcal{S}_{rem}$

$\mathcal{S} \leftarrow [(0,0), (1,0), \cdots, (N_{out}-1, 0)]$;

$\mathcal{S}_{rem} \leftarrow \mathcal{S}_{rem}.zipWithIndex.flatten$; // add field _.2, the partition id.

OSort($\mathcal{S}_{rem}$, _.1); // in ascending order, sorted $\perp$ to the end.

for $i \leftarrow 0$ **to** $|\mathcal{S}_{rem}|$ **do**

- $k \leftarrow i \mod N_{out}$;
- cmov($\mathcal{S}_{rem}[i].1 \neq \perp$, $\mathcal{S}[k].1$, $\mathcal{S}[k].1 + \mathcal{S}_{rem}[i].1$);
- cmov($\mathcal{S}_{rem}[i].1 \neq \perp$, $\mathcal{S}_{rem}[i].1$, k); // reinterpret the _.1 field to record the final assignment result.

OSort($\mathcal{S}_{rem}$, (_.2, _.0));

$\mathcal{S}_{rem} \leftarrow$ Split $\mathcal{S}_{rem}$ by different _.2 and strip field _.2;

return $\mathcal{S}$, $\mathcal{S}_{rem}$;

Algorithm 13: Oblivious Join: Stage 3

function OJoinStage3($\mathcal{S}_r$, $\mathcal{S}_o$, l_r, l_o, $\mathcal{S}_{rem}$):

Input: Received and owned column sum info $\mathcal{S}_r$ and $\mathcal{S}_o$, last bin location l_r and l_o, remaining bin info $\mathcal{S}_{rem}$

Output: To-be-sent and owned column sum info $\mathcal{S}_r$ and $\mathcal{S}_o$, updated bin location l_o

$N_{out} \leftarrow |\mathcal{S}_o|$;

OSort($\mathcal{S}_r$, _.1); // descending.

OSort($\mathcal{S}_o$, _.1); // ascending.

$\mathcal{S}_r.1 \leftarrow \mathcal{S}_o.1 + \mathcal{S}_r.1$; // vectorized add.

$\mathcal{S}_o.1 \leftarrow \mathcal{S}_r.0$; // record the reassignment result.

OSort($\mathcal{S}_o$, _.0); // restore the original order.

/* Adjust partition number for the last bin. */

$l \leftarrow l_o$;

foreach $s \in \mathcal{S}_o$ **do**

- $c \leftarrow l \geq 0 \wedge l = s.0$;
- cmov(c, l_o, $s.1$); // if the last bin is in regular bins.

cmov($l = -2$, l_o, $\mathcal{S}_{rem}[0].1$); // if the last bin is in remaining bins.

cmov($l = -1$, l_o, l_r);

return $\mathcal{S}_r$, $\mathcal{S}_o$, l_o;

Algorithm 14: Coordinate Bin Number

function CoordBinNum($\mathcal{P}$, cap, j, $\mathcal{I}$):

Input: Partition after aggregate $\mathcal{P}$, bin capacity cap, partition id j, last bin information of all partitions $\mathcal{I}$

Output: Partition after adjusting bin number $\mathcal{P}$

/* Decide the first and the last bin number of this partition. */

$bin\_num \leftarrow \mathcal{I}[0].0$;

$bin\_size \leftarrow \mathcal{I}[0].1$;

$n \leftarrow bin\_num + 1$;

$n_{last} \leftarrow 0$;

for $i \leftarrow 1$ **to** $|\mathcal{I}| - 1$ **do**

- /* Try to place in the old bin, the bin capacity is cap. */
- $cur\_bin\_num \leftarrow \mathcal{I}[i].0$;
- $num, size \leftarrow$ NextFit(cap, $true$, $\mathcal{I}[i].1$, bin_num, bin_size, $n + cur\_bin\_num$);
- **if** $i = j$ **then**
  - $n_{last} \leftarrow n$;
- cmov($bin\_size + \mathcal{I}[i].1 > cap$, n, $n+1$); // as a new bin is created.
- $n \leftarrow n + cur\_bin\_num$;
- $bin\_num, bin\_size \leftarrow num, size$;
- $\mathcal{I}[i].0 \leftarrow bin\_num$;

/* Assign new bin number according to old bin number. */

$tmp\_num \leftarrow \mathcal{P}[0].b$;

for $i \leftarrow 0$ **to** $|\mathcal{P}| - 1$ **do**

- $c \leftarrow \mathcal{P}[i].b \neq tmp\_num$;
- cmov(c, $\mathcal{P}[i].b$, $(n_{last} + \mathcal{P}[i].b)$);
- cmov($\neg c$, $\mathcal{P}[i].b$, $\mathcal{I}[j].0$);

return $\mathcal{P}$;

Algorithm 16: Oblivious Join: Stage 5

```
function OJoinStage5(𝒫, ℬ, β, N_in, N_out):
    Input: Partition 𝒫, buckets with aggregate information ℬ,
           total number of items β, # of nodes before group by
           N_in, # of nodes after group by N_out
    Output: Buckets with original data ℬ′
    for t ← 0 to 1 do
        𝒫[t] ← PatchPartNum(𝒫[t], ℬ[t], N_in);
    compute the padding length l;
    ℬ′ ← BuildBuckets(𝒫.flatten(), l, N_out)
    return ℬ′;
function PatchPartNum(𝒫, ℬ, N_in):
    𝒟 ← OMerge(ℬ, _.l);
    for i ← 0 to |𝒟| − 1 do
        𝒫[i].b ← 𝒟[i].b;
    /* 𝒫 has three fields: k, v, b. The last one in the items with the same key
       always has the accurate new partition number. Use it to fix others
       with the same key. */
    for i ← |𝒫| − 2 to 0 do
        c ← 𝒫[i].k = 𝒫[i + 1].k;
        cmov(c, 𝒫[i].b, 𝒫[i + 1].b);
    return 𝒫;
```

Algorithm 17: Oblivious Join: Stage 6

```
function OJoinStage6(ℬ, len):
    Input: Buckets collected from all nodes ℬ, pad length for join
           result len
    Output: Partition after join 𝒫
    /* Note that each bucket is already sorted by k */
    𝒫 ← OMerge(ℬ, (¬is_valid(v), _.k)) ;                    // merge by k
    𝒫 ← Apply single-node oblivious join in [1] on 𝒫. During the
      oblivious expansion, pad both to len;
    return 𝒫;
```

Algorithm 15: Oblivious Join: Stage 4

```
function OJoinStage4(𝒫, 𝒟, 𝒮, l, 𝒮_rem):
    Input: Partition after aggregate 𝒫, intermediate result for sum
           𝒟, product sum of columns 𝒮, last bin location l, last
           row of bins 𝒮_rem
    Output: Buckets ℬ
    /* Prepare the patch of new partition numbers. */
    N_out ← |𝒮|;
    cur ← 0;
    for i ← 0 to |𝒟| − 1 do
        cmov(𝒟[i].1 ≠ ⊥, cur, (cur + 1) mod N_out);
    tmp ← 𝒮_rem;      // 𝒮_rem is already sorted by the index (mod N_out)
    for i ← ⌈|𝒟|/N_out⌉ − 1to 0 do
        chunk ← 𝒟[i × N_out : (i + 1) × N_out];            // reference
        buf ← chunk;                                          // a copy.
        should_update ← whether _.1 ≠ ⊥ is for all in chunk;
        last ← cur;
        for j ← N_out − 1 to 0 do
            c ← chunk[j].1 ≠ ⊥ ∧ chunk[j].1 < last;
            cmov(c, chunk[j].2, tmp[j].1);          // assign partition
              number to current chunk
            cmov(c, cur, cur − 1);
            buf[j].1 ← j;                       // record the original order.
        OSort(buf, _.2); // By old column number. If it contains dummy,
          it will not be used. So no OPlace is needed.
        buf[..].2 ← 𝒮[..].1;                     // vectorized assignment.
        OSort(buf, _.1);                          // restore the previous order.
        cmov(cur = 0 ∧ should_update, cur, N_out);
        for j ← N_out − 1 to 0 do
            cmov(should_update, tmp[j].1, buf[j].2);   // update
              tmp for patching next chunk.
            c ← chunk[j].1 ≠ ⊥ ∧ chunk[j].1 ≥ last;
            cmov(c, chunk[j].2, tmp[j].1);          // assign partition
              number to next chunk
            cmov(c, cur, cur − 1);
        OSort(chunk, _.0);                              // the original order.
    /* Assign new partition number. */
    𝒫[0].b ← l;
    for i ← 1 to |𝒫| − 1 do
        cmov(𝒟[i].1 ≠ ⊥, 𝒫[i].b, 𝒟[i].2);
        cmov(𝒟[i].1 = ⊥, 𝒫[i].b, 𝒫[i − 1].b);
    /* Prepare to patch new partition number back to original items. */
    OSort(𝒫, (_.t, _.i, _.l)) ;                          // sort by (t, i, l)
    ℬ ← Split 𝒫 by different t, i and strip fields t, i;
    return ℬ;
```

REFERENCES

[1] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2132–2145. https://doi.org/10.14778/3407790.3407814