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
Shanghai Artificial Intelligence Lab
Shanghai Qi Zhi Institute
gaomy@tsinghua.edu.cn

ABSTRACT

Cloud systems are now a prevalent platform to host large-scale big-data analytics applications such as machine learning and relational database. However, data privacy remains as a critical concern for public cloud systems. Existing trusted hardware could provide an isolated execution domain on an untrusted platform, but also suffers from access-pattern-based side channels at various levels including memory, disks, and networking. Oblivious algorithms can address these vulnerabilities by hiding the program data access patterns. Unfortunately, current oblivious algorithms for data analytics are limited to single-machine execution, only support simple operations, and/or suffer from significant performance overheads due to the use of expensive global sort and excessive data padding.

In this work, we propose SODA, a set of efficient and oblivious algorithms for distributed data analytics operators, including filter, aggregate, and binary equi-join. To improve performance, SODA completely avoids the expensive oblivious global sort primitive, and minimizes the data padding overheads. SODA makes use of low-cost (pseudo-)random communication instead of expensive global sort to ensure uniform data traffic in oblivious filter and aggregate. It also adopts a novel two-level bin-packing approach in oblivious join to alleviate both input redistribution and join product skewness, thus minimizing necessary data padding. Compared to the state-of-the-art system, SODA not only extends the functionality but also improves the performance. It achieves 1.1× to 14.6× speedups on complex multi-operator data analytics workloads.

PVLDB Reference Format:
Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. SODA: A Set of Fast Oblivious Algorithms in Distributed Secure Data Analytics. PVLDB, 16(7): 1671 - 1684, 2023.
doi:10.14778/3587136.3587142

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tsinghua-ideal/flare/tree/oblivious_soda.

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587142

1 INTRODUCTION

Cloud-based data analytics now become prevalent. The cloud offers great computation capabilities at relatively low prices to process vast amounts of data, using both high-performance hardware resources and highly optimized distributed data analytic frameworks like Spark [68]. Many entities, both large companies and individual users, are willing to outsource their data and computation tasks to the cloud, including machine learning [1, 45], graph processing [25], and relational data analytics [7]. However, cloud computing also brings serious challenges of data privacy, where users' sensitive data might be eavesdropped or tampered with by malicious attackers on such publicly shared platforms.

Cloud platforms are currently adopting various system-level solutions to address security issues. Hardware-based trusted execution environments (TEEs), such as Intel SGX [5, 29, 44], ARM TrustZone [3], and AMD SEV [57], have been proposed for years, and also seen real-world deployment in the cloud. They aim to provide a secure and isolated domain for sensitive computations on the untrusted cloud platform. Unfortunately, TEEs are not a panacea. Various side channels have been demonstrated in existing TEEs [22, 27, 35, 42, 55, 61, 67, 69], allowing attackers to bypass the protection mechanisms. Many known side channels can be attributed to data access pattern leakage in the memory and disk hierarchy [27, 61, 67]. Furthermore, TEEs are only limited to a single machine. With large-scale distributed execution, the network traffic among server nodes also becomes a source of leakage [49].

Protecting against such memory, disk, and networking access pattern vulnerabilities requires the system to perform *oblivious* execution, meaning that the externally visible behaviors (e.g., memory accesses and networking transfers) of the program should be deterministic and independent of the sensitive data. Generic cryptographic protocols like oblivious RAM (ORAM) can achieve this goal and prevent access pattern leakage [21, 23, 24, 50, 52, 60], but they usually incur substantial performance overheads. Customized oblivious algorithms are tailored to a particular application domain and thus offer better performance. We focus on oblivious algorithms for data analytics in this work. However, most existing proposals only considered single-node execution [6, 12, 19, 34, 41] and cannot be securely generalized to distributed scenarios. The few distributed solutions either only supported a limited set of simple operations [18, 49], or incurred excessive performance cost [70].

For example, Opaque [70], the state-of-the-art distributed data analytics framework, relied on a general but expensive primitive, oblivious global sort, to construct oblivious filter, aggregate, and join operators. The specific implementation of global sort, namely column sort [36], required four rounds of local oblivious sort on each node, interleaved with four rounds of global data shuffle across all nodes. Both the computation and communication cost was significant. In addition, it only supported primary-foreign join where one of the two input tables does not contain duplicated keys. The more general equi-join cannot be executed securely on this system. A straightforward extension to support equi-join would require a significant amount of data padding to hide the data distribution information (Section 3). Therefore, better oblivious algorithms must effectively alleviate the *global sort* cost and the *padding* overheads.

In this paper, we propose SODA, a set of efficient and oblivious algorithms for distributed data analytics operators, including filter, aggregate, and binary equi-join. SODA completely avoids the expensive oblivious global sort primitive and minimizes the data padding overheads to improve performance. SODA achieves its efficiency with two key insights. First, we leverage (*pseudo-*)*random communication*, where the records in an input partition are sent to each destination node in a (pseudo-)random manner. Such low-cost communication can be made fully oblivious by applying a small amount of padding (e.g., 10%) to ensure the exact same data volume is sent to each node under negligible failure rates (e.g., 2×10^{-57}). Therefore, in **oblivious filter**, SODA uses such random communication to distribute both the valid and invalid records evenly across all nodes before local filtering, in order to avoid per-partition output size leakage. In **oblivious aggregate**, SODA makes use of the other records that do not store the aggregated results, and changes their keys into randomly assigned dummy keys, in order to apply (pseudo-)random communication to hide data distribution, while still ensuring correct aggregate results.

Second, to also support efficient **oblivious join**, SODA utilizes the aforementioned aggregate operator to first find out the overall data distribution, and then applies a novel *two-level bin-packing assignment* to decide how to group the records with the same key across the server nodes. The two-level bin-packing is conducted in an oblivious manner and ensures that both the two input tables and the output join result table are evenly spread without exposing any input redistribution and join product skewness of the data records.

We evaluate SODA on real-world benchmarks and datasets, including BigDataBench [64], diverse join queries on TPC-H and social graphs [12], and other complex workloads consisting of multiple operators. On single-operator benchmarks, SODA oblivious filter and aggregate outperform the Opaque-like baseline [70] by about 2×, and oblivious join achieves 1.1× to 18× speedups depending on the join type and data distribution. For complex multi-operator benchmarks, SODA offers performance benefits ranging from 1.1× to 14.6×. We have open sourced SODA at https://github.com/tsinghua-ideal/flare/tree/oblivious_soda.

2 BACKGROUND

2.1 Trusted Execution Environments

A promising hardware technique used nowadays for secure data processing is *trusted execution environments* (TEEs), such as Intel SGX [5, 29, 31, 44], Intel TDX [32], ARM TrustZone [3], and AMD SEV [57]. TEEs can create a trusted domain (called an *enclave*) on an untrusted platform and allow computing directly on plaintext data in it. The hardware prohibits illegal access to the trusted memory from the untrusted part, guaranteeing isolation. The integrity (i.e., correctness) of the code in the enclave is ensured through attestation [5]. When leaving the processor, data in the trusted domain are automatically encrypted and authenticated by hardware, e.g., a Memory Encryption Engine (MEE) [28] or a Total Encryption Engine (TME) [30] in SGX, so they cannot be eavesdropped or tampered with when stored in the external memory.

However, the security of existing TEEs has been questioned due to various side-channel vulnerabilities at the microarchitectural and physical levels [22, 27, 35, 42, 55, 61, 67, 69], among which an important one is related to the leakage of access patterns. In other words, TEEs do not hide the addresses of sensitive data accesses. Such exposed access traces may leak certain information. Consequently, even inside enclaves, we still need *oblivious* techniques, such as oblivious RAM (ORAM) [52, 60] or customized oblivious algorithms [6, 9]. Many existing designs [19, 46, 53, 70] on TEEs achieve such a *doubly oblivious* property, i.e., ensuring obliviousness in both the trusted and untrusted domains. They either integrate the generic ORAM protocols into TEEs, or utilize a limited space (e.g., a few MBs) of hardware oblivious memory (OM) [20, 59]. In this work, we design customized oblivious algorithms that are more efficient than ORAM and do not rely on hardware OM whose implementation requires subtle hardware modifications.

2.2 Oblivious Distributed Data Analytics

A non-oblivious program may expose the data access patterns at multiple levels, including memory, disks, and networking. Most existing oblivious data analytics algorithms [6, 12, 19, 34, 41, 46] only protect access patterns on a single machine, where the main consideration is at the memory and disk levels. On distributed platforms such as MapReduce [16], Hadoop [16], and Spark [68], data traffic across multiple machines through network becomes another attack surface. In fact, because the network traffic is visible to the cloud provider as well as other cloud tenants even if they are *not* on the same physical machine, it is a more vulnerable threat.

Ohrimenko et al. [49] were the first to realize obliviousness in a distributed MapRduce system. They proposed two provably secure and practical solutions, corresponding to two levels of security definitions. The weaker solution prevented attackers from tracing the traffic between individual mappers and reducers, by conducting secure shuffle on all the key-value pairs. Nevertheless, attackers could still obverse the distinct intermediate data volume of each mapper/reducer, and thus learn the distribution of keys. To further protect against this issue, the stronger solution used offline preprocessing to achieve a uniform distribution for the intermediate traffic, i.e., the data volume from each mapper to each reducer was the same. However, it still made a tradeoff between performance and security, by allowing to leak *the frequency of the most popular key* in the dataset [49]. Otherwise, we would need to pad the traffic between each mapper and reducer to the worst case, which could be the entire data volume if all records have the same key, obviously diminishing any advantage of distributed execution.

Opaque [70] is the state-of-the-art oblivious distributed computing platform based on Spark [68]. Spark supports distributed execution through an important abstraction named resilient distributed datasets (RDDs). An RDD consists of several partitions. During execution, the scheduler in Spark dispatches computing tasks to multiple worker nodes, where each node typically processes one RDD partition in memory. Spark supports flexible query types including filter, aggregate, and primary-foreign join. Opaque proposed distributed oblivious algorithms for these operators, implemented using TEEs like Intel SGX. Its main building block was a relatively expensive, oblivious global sort primitive. We describe the detailed design of Opaque in Section 3 as our baseline.

2.3 Threat Model

In this work, we consider a scenario where a client would like to perform data analytics queries on the outsourced data in a remote and untrusted server cluster with multiple server machines. The dataset has been securely (e.g., with encryption and authentication) uploaded and distributed across the cluster. The server machines are untrusted, and a powerful adversary can control the whole software stack, including the operating systems (OSes) and the hypervisors. We assume the hardware processors on the servers are trusted and equipped with hardware TEEs. The adversary still controls the software running on the processors, but the code inside the enclaves can be verified through attestation to ensure integrity. The adversary can snoop or tamper with everything outside the processors, including data transfers at the memory, disks, and network devices. In particular, access address traces and network traffic are exposed, from which the adversary can infer secret information even if data encryption has been applied [49].

Following common practice, we do not consider denial-of-service attacks, as well as physical attacks that exploit other side channels such as electromagnetic [22], thermal [42], and power [69]. We do *not* assume the availability of hardware oblivious memory in our algorithm designs. It is possible to incorporate hardware oblivious memory into our algorithms to further improve the performance, which we leave as future work.

2.4 Security Definition

We adopt encryption and authentication techniques to ensure confidentiality, integrity, and freshness, both during execution and when data are statically stored, following common practice in previous work [49, 70]. We mainly focus on the obliviousness issue of executing a set of operators on the distributed dataset, including filter, aggregate, and binary equi-join. Below we clearly state the definition of obliviousness in our work, which is extended from Opaque [70] by also integrating the concepts in Ohrimenko et al. [49]. The main improvement beyond Opaque is to cover general binary equi-join since Opaque only supports primary-foreign join.

Generally speaking, to ensure obliviousness, we need to restrict what the adversary could observe. Similar to [19, 49], we only allow for minimum leakage of the input and output dataset sizes, which are usually considered harmless and inevitable because these data must be transferred in and out of the system. We denote the input dataset (or table) as D, and the output result as R, which is produced by running an oblivious program P on D, i.e., $R = P(D)$. P can be viewed as a physical plan of the query after query optimization. We assume that each attribute value in a table has a fixed size; otherwise, it is padded to a constant size. For filter and aggregate, the access patterns only depend on

(1) The specific algorithm P being computed;
(2) The input dataset size (the number of table rows) β of D;
(3) The output dataset size (the number of table rows) γ of R;
(4) The schemas of D and R, respectively denoted as $\mathsf{Schema}(D)$ and $\mathsf{Schema}(R)$, including the table names, the attribute names, and the attribute value sizes;
(5) The number of partitions (the number of machines) N.

This set of information leakage is the same as assumed in Opaque [70]. For binary equi-join, the access patterns additionally depend on

(6) The number of the most popular key α in the input D [49].

Note that binary equi-join involves two input datasets A and B, so the above leakage on the input dataset D includes both tables.

More formally, we define Trace to be the execution trace of a sensitive operator. Obliviousness means that there exists a probabilistic polynomial-time algorithm Sim (a simulator) such that

$$\mathsf{Sim}(\mathsf{params}) \cong \mathsf{Trace}(\mathsf{params}, D)$$

where $\mathsf{params} = (P, \beta, \mathsf{Schema}(D), \gamma, \mathsf{Schema}(P(D)), [\alpha])$; α is only included for join. $\cong$ means the two sides are indistinguishable. We remark we allow the *full* input/output dataset size (the number of rows in a full table) to be revealed to the adversary, but not that in an individual *partition* if the partition size is related to the sensitive content. For example, when using hash-based partitioning, the individual partition size reveals the record key distribution. We assume the initial partitioning scheme already satisfies this requirement and evenly splits data onto the servers. The oblivious operator execution ensures the output partitioning is still irrelevant to record content, and the result can be used as the input for another operator.

3 BASELINE AND CHALLENGES

In this section, we describe our baseline design in detail. The baseline is an extension of Opaque [70]. Opaque supported a set of distributed oblivious operators. It relied on a key primitive, column sort [36], which realized an oblivious global sort across all server nodes. One column sort requires 4 rounds of local oblivious sort (e.g., bitonic sort [48]) on each node, interleaved with 4 rounds of global shuffle on the entire dataset across all nodes. Each local oblivious sort works on the local partition (called a column). Each global shuffle exchanges data among columns in a deterministic way. As long as the relationship between the number of partitions s and the number of records per partition r satisfies $r \geq 2(s-1)^2$ [36], the number of rounds is constant for arbitrary data size. All the oblivious operators in Opaque invoked such expensive column sort 1 or 2 times, which incurred high costs.

Our baseline directly uses the oblivious filter and aggregate designs in Opaque [70]. More concretely, for filter, to avoid revealing which rows are filtered and how many rows in a partition are filtered, the baseline firstly marks the to-be-filtered rows with a sequential scan and performs a global oblivious sort to put these rows together at the end of the table. Then, removing them only leaks the total number of rows in the output result, satisfying our security definition in Section 2.4. For aggregate, we need to prevent

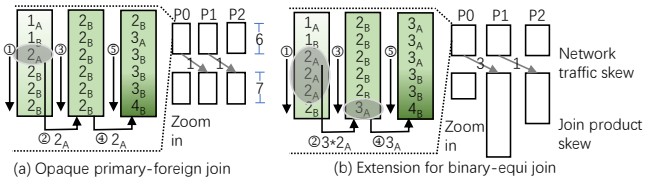


Figure 1: An example showing the communication in the Opaque primary-foreign join, and the extended design of generic equi-join in the baseline.

the adversary from tracking which output partition each individual input record goes to. A global oblivious sort is first performed, thus the records with the same group key are located contiguously from the view of the entire table. Because the table is physically separated into several partitions across multiple nodes, some groups may be split across partitions. Therefore, after performing scan-based local aggregate within each partition, the nodes need to further communicate with each other to obtain the partially aggregated values and finally derive the correct aggregate results. To ensure obliviousness, this communication may involve dummy records, which need to be discarded by a final filter.

Opaque only supports primary-foreign join [12, 34], where one of the involved tables is required to be the primary key table, and each join key can at most have a single record in it (there is no restriction on the other foreign key table). Opaque first unions the two tables, and conducts a global sort on the join key. The sort ensures that for the same join key, the single primary table record is always in front of those foreign table records. Then, a per-partition scan propagates the primary table record and uses it to join with every foreign table record in each join group. The partition boundaries require special handling similar to the aggregate case, i.e., each node receives the last primary table record from the previous node.

Our baseline extends Opaque to enable more general binary equi-join using straightforward padding. The first change is in the boundary processing, because more than one records may need to be communicated from one node to another. For the example in Figure 1, partition 0 needs to obliviously pass 3 records with key 2_A of table A to partition 1 for a binary equi-join. Therefore, we need to pad the number of records sent and received at each node to the same public volume. Instead of padding to the total number of rows in a table to accommodate the worst case, our security definition in Section 2.4 allows for revealing the maximum frequency of the keys in a table, i.e., α, which is the upper bound of communication traffic per node. However, obliviously extracting those records would need another expensive local oblivious sort. Our baseline design instead uses a simpler approach that trades some extra communication traffic for less computation, by directly sending the last $\alpha_A + \alpha_B$ records to the next node. The boundary keys are guaranteed to be within these records.

The second issue is that the local processing within each partition is also complicated by the many-to-many record relationship between the two tables, and the single-scan method in Opaque cannot be used. We could apply a generic single-node oblivious equi-join algorithm [12, 34], but we still need one more padding to hide the output partition size on each node. This is because the different join degrees of the groups in each partition may lead to output size skewness, i.e., the numbers of records after join are not equal across the partitions. This skew allows the attacker to learn some distribution information of the join key. We again calculate the worst case for one partition that leads to the maximum join output size. This happens when all join groups are maximally sized (with α_A and α_B) except for the last remainder group.[1] In this scenario, the maximum output size is $\min(n', \gamma)$ where

$$n' = \left\lfloor \frac{n}{\alpha_A + \alpha_B} \right\rfloor \cdot (\alpha_A \alpha_B) + \left(\frac{n \bmod (\alpha_A + \alpha_B)}{2} \right)^2$$

Here n is the partition size after receiving the records of previous partitions but before local join. The first term is the output size for the maximum-sized join groups. The second term accounts for the remainder group.[2] Roughly speaking, this padding introduces an $O(\alpha)$ (if dominated by n') or N (number of partitions, if dominated by γ) multiplicative factor to the partition size. The large output size significantly increases the overheads of the final global sort for filtering dummy records.

Summary of challenges. Our baseline design described in this section suffers from two key inefficiencies. First, all the operators heavily use the *expensive oblivious global sort* primitive, which itself requires multiple rounds of global communication and local processing. Both the local and global data movements become particularly costly when the table has many attributes besides the key, which only add to the total data volume without impacting the sort order. Second, for oblivious join, there is *significant data padding* applied throughout the processing, introducing a large amount of dummy records in the intermediate results. Excessive dummy records would greatly slow down many computations whose complexity is super-linear with the data size. In particular, the aforementioned global sort becomes even more expensive. We would like to minimize the padding volume while still ensuring the security definition in Section 2.4. To address these issues, we next propose more efficient oblivious operators for distributed data analytics.

4 DESIGN

We propose SODA, a set of efficient and oblivious algorithms for distributed data analytics operators, including filter, aggregate, and binary equi-join. SODA does not rely on the expensive oblivious global sort primitive and minimizes the data padding overheads to improve performance. We first describe in Section 4.1 two basic primitives used by SODA, oblivious sort and distribute. Section 4.2 then introduces (pseudo-)random communication, the key insight we leverage to realize efficient oblivious global communication. Finally, we elaborate the detailed algorithms for oblivious filter, aggregate, and binary equi-join, respectively in Sections 4.3 to 4.5.

4.1 Basic Primitives

SODA makes use of two basic primitives, oblivious sort and distribute, to realize high-level oblivious operators. Both primitives are local to one partition at a node, without global data movements.

[1] Assume there are two groups x and y that are not maximally sized. They have $x_A + y_A$ records from table A and $x_B + y_B$ records from table B. This case does not result in the maximum output size because $x_A x_B + y_A y_B < \alpha_B \alpha_B + (x_A + y_A - \alpha_A)(x_B + y_B - \alpha_B)$.
[2] The remainder group has in total $x_A + x_B = n \bmod (\alpha_A + \alpha_B)$ records from the two tables A and B, so the output size $x_A x_B \le ((x_A + x_B)/2)^2$.

Single-node oblivious sort has been well studied [2, 8, 26, 48]. We choose bitonic sort [48] as our `OSort` implementation, which has a time complexity of $O(n \log^2 n)$. Although there exist asymptotically better constructions with $O(n \log n)$ [8], bitonic sort is practically more efficient with smaller constant factors, and is more friendly to parallel processing on modern processors.

A special case of sort is to merge multiple sorted sequences. It is not necessary to perform the full bitonic sort procedure. We provide a more efficient variant `OMerge`. Recall that bitonic sort runs on arbitrary-size sequences (not just powers of two) in a recursive manner. If we concatenate the input sorted sequences into one, we can view each sub-sequence as already sorted by recursive small bitonic sort, and then only need to apply the remaining steps. We also use `OMerge` whenever possible in our baseline. It can replace three out of the four local sort rounds in one column sort.

Krastnikov et al. [34] proposed an "oblivious distribute" algorithm. Given an input data array $\mathcal{D} = \{x_1, \ldots, x_n\}$, m out of the n elements are real, while the others are dummy. The expected location of each real x_i, denoted as $f(x_i) \in \{1, \ldots, n\}$, is in the ascending order. If $\forall i, f(x_i) \geq i$, `ODistribute` puts each real element x_i to the location $f(x_i)$, with a time complexity $O(n \log n)$.

4.2 (Pseudo-)Random Communication

The major difficulty of designing SODA is how to realize oblivious data communication without relying on the expensive global sort and with minimum data padding. We observe that, if we could ensure *each node sends the same amount of data to all receiving nodes* during global communication, then the communication is trivially oblivious. However, the traffic must follow certain constraints. For example, aggregate and join require the records with the same key, a.k.a., those in the same aggregate/join group, to be sent to the same node for processing. With real datasets, data skewness would cause communication imbalance and thus violate obliviousness. Prior systems [49, 66, 70] either used oblivious global sort to collect the groups instead of direct communication, or applied excessive padding to the worst-case size, both incurring high overheads.

Our key insight is that *(pseudo-)random communication* among the nodes, in which the destination of each record is chosen uniformly at (pseudo-)random, is *nearly* oblivious. More specifically, when we need to globally shuffle data among all the N nodes, the data partition $\mathcal{P}$ at each node is split into N buckets, each of which is then sent to one other node. The bucket ID of each record in $\mathcal{P}$ should be assigned in a (pseudo-)random manner. Then the resultant N buckets would have nearly the same size, representing a nearly uniform communication volume to other nodes.

However, a simple (pseudo-)random bucket assignment does not guarantee full obliviousness. The small size differences may leak information in certain scenarios, in particular when there are both valid and invalid records. Similar to previous systems [49, 66, 70], the data partitions in SODA also contain invalid records, either those marked to be filtered by a filter operator, or the dummy records used for the padding purpose. Assume an example where two nodes 0 and 1 are exchanging data, and each node sends slightly more data to itself than to the other, i.e., traffic $0 \to 0$ is larger than $0 \to 1$, and traffic $1 \to 1$ is larger than $1 \to 0$. If the final data sizes after filtering out invalid records at the two nodes are not the same, e.g., node 0 has more final valid records than node 1, then we can infer that originally node 0 is likely to have more valid records. In the extreme case, if both nodes have only 1 record that is assigned to local, and one is valid and the other is invalid, then we surely know the original per-partition amount of filtering, which does not satisfy our security requirements in Section 2.4.

Algorithm 1: (Pseudo-)Random Communication Buckets

```
1  function BuildBuckets(P, N):
       Input: Data partition P with (pseudo-)randomly assigned
              bucket field per record, total number of partitions N.
       Output: N buckets B.
2      Compute the padding size l ← |P|/N + k√(|P|/N);
3      OSort(P, (_.bucket, _.valid, _.key));
       /* Assign locations in buckets. */
4      b ← ⊥; p ← ⊥;
5      foreach d ∈ P do
6          c ← d.bucket ≠ b ;                    // encounter a new bucket?
7          b ← d.bucket;
8          cmov(c, p, l × b) ;                   // starting location of a new bucket.
9          d.pos ← p; p ← p + 1;
       /* Place records to assigned locations. */
10     Pad P to length l × N;
11     ODistribute(P, _.pos) ;   // invalid records at end of each bucket.
12     B ← Chunk P into N buckets, each of size l;
13     return B;
```

Therefore, to ensure obliviousness, the N buckets built from one partition must have exactly the same size. Given that we are randomly assigning $|\mathcal{P}|$ records to N buckets, we set the padded bucket size to $l = |\mathcal{P}|/N + k\sqrt{|\mathcal{P}|/N}$, where the second term represents the padding overhead. This leads to an exponentially decreasing failure rate of $N \exp(-k^2/3)$ according to the Chernoff bound (Chapter 4.2 in [47]). k is a tunable parameter that trades off between padding size and failure rate. In SODA we set $k = 20$ to get a failure probability of $2{\times}10^{-57}$ when $N = 16$ (Section 6.3). This negligible probability is lower than the typical distributed system failure rate, and is not a concern in practice; we could simply re-run the program.

In summary, we construct a sub-routine `BuildBuckets` in SODA, as in Algorithm 1. The destination bucket of each record should be assigned in a (pseudo-)random manner before invoking this sub-routine. We sort the partition into valid and invalid records assigned to different buckets and determine the exact location in the padded destination bucket for each record. Note that we use conditional move (`cmov`) instructions to ensure oblivious processing. Finally, we use `ODistribute` to obliviously form the buckets.

Next we describe how to turn data-dependent communication in aggregate and join into (pseudo-)random communication.

4.3 Oblivious Filter

SODA has two types of oblivious filter operators, for local and global, respectively. The local filter is meant to be used as a sub-routine only invoked by high-level operators; it does not fully satisfy our security requirements in Section 2.4.

Local filter. A simple implementation is to locally and obliviously sort the to-be-filtered records to the end of the local partition

Algorithm 2: Oblivious Local Filter

```
1  function OLocalFilter(𝒟):
       Input: Input array 𝒟 to be filtered according to field valid.
       Output: Output array ℛ after filtering.
       /* Count and annotate the valid records. */
2      cnt ← 0;
3      foreach d ∈ 𝒟 do
4          d.pos ← ⊥;                                   // add new field pos.
5          cmov(d.valid, d.pos, cnt);
6          cmov(d.valid, cnt, cnt + 1);
7      ℛ ← [⊥, ⊥, ..., ⊥]; start ← 0 ;                  // |ℛ| = cnt.
8      for i ← 0 to ⌈|𝒟|/cnt⌉ − 1 do
9          chunk ← 𝒟[i × cnt : (i + 1) × cnt];
           /* Assign locations to invalid records, to take the space before the
              valid records in the chunk. */
10         cur ← 0; last ← start − 1;
11         foreach d ∈ chunk do
12             c ← d.pos ≠ ⊥ ;                          // whether it is a valid record.
13             cmov(c, last, d.pos) ;                   // update location of last valid record.
14             c ← (cur < start) ∧ (d.pos = ⊥);
15             cmov(c, d.pos, cur);
16             cmov(c, cur, cur + 1);
17         start ← last + 1;
           /* Obliviously sort the chunk; valid records are in their final
              locations, with invalid records before and after. */
18         chunk ← OSort(chunk, _.pos);
           /* Obliviously copy to the output array. */
19         for j ← 0 to |ℛ| do
20             c ← (ℛ[j] = ⊥) ∧ (chunk[j].pos ≠ ⊥);
21             cmov(c, ℛ[j], chunk[j]);
22     return ℛ;
```

and then remove them. The sort overheads could be significant for large partitions, especially when there are only a few valid records that should be kept. We thus propose an optimized design. As shown in Algorithm 2 and Figure 2, SODA first counts and annotates all the valid records with their final locations in the output array (the new pos field). The input array is then chunked into the size of the output array. In each chunk, the valid records are sorted to the locations according to the annotation, with invalid records filling in the space before and after them. Now the position of each valid record in its current chunk is the same as its expected position in the final output array. Finally, we obliviously copy the valid records in the chunk to the output array while preserving their relative positions within the chunk.

Global filter. We cannot directly use the local filter to remove invalid records at each node independently; otherwise, it leaks per-partition record information, which is prohibited in our security model. Nevertheless, our key insight is that, if we first *randomly distribute both valid and invalid records across all nodes*, then the distribution of valid records, i.e., the number of valid records at each node, is irrelevant to the data content and thus does not leak sensitive information. Now we could apply local filter operators to all partitions in parallel. Such random data distribution can be directly realized by our random communication in Section 4.2.

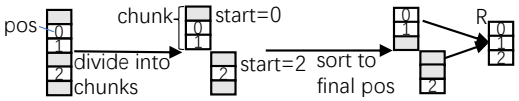


Figure 2: Overall flow of SODA oblivious local filter. Records in dark are invalid. The key step shown in the middle sorts the valid records in each chunk to their final positions in the result, e.g., record 0 at position 0 and record 1 at position 1 in their own chunks, respectively.

Consequently, we propose our optimized global filter, which is divided into two stages with data communication in between. In stage 1, we assign to each record in the local partition a randomly chosen destination bucket ID between 0 and N, where N is the number of output partitions. Then we use `BuildBuckets` in Algorithm 1 to construct the buckets and conduct a random communication. In stage 2, each node collects all the buckets sent to it and uses the above oblivious local filter operator to remove the invalid records.

Efficiency. Our oblivious filter operator does not use any expensive global sort. It only uses one round of (pseudo-)random communication, which greatly improves upon our baseline which uses four rounds of data shuffling. The cost of local processing is also smaller than the four `OSort`/`OMerge` calls in the baseline. It now includes several sequential scans, two `OSort`/`OMerge` calls, and one `ODistribute` [40]. This leads to $O(n \log^2 n)$ time complexity and $O(n)$ space complexity, determined by the `OSort` primitive.

4.4 Oblivious Aggregate

To avoid expensive global sort, we design the oblivious aggregate in SODA based on hash-based partitioning. Naive hash-based partitioning would leak the destination node of each record as well as the size of each aggregate group (i.e., records of the same key). We leverage the aggregate property to avoid these issues. For all the records belonging to one group, *only one record is responsible for holding the aggregate value, while the others can be changed to dummy records whose keys are freely set.* Therefore, after local aggregate, we turn these other records into random dummy ones that contain different keys from each other and also different from any real keys. With a uniform hash function applied on these different keys to derive their destination partitions, the overall traffic satisfies our pseudo-random communication requirement, and thus hides the sensitive data distribution. Also, because the destination is fully determined by the key, all records with the same real key will still be directed to the same node, ensuring correct aggregate results.

Specifically, the SODA oblivious aggregate operator contains two stages, as illustrated in Figure 3. In stage 1 as Algorithm 3, each node independently aggregates the local records in its partition. This can be done by first locally sorting all the records to form consecutive aggregate groups, and then doing a sequential scan to aggregate each group. The last record in each group holds the (partially) aggregated result and is valid, while the others are marked invalid. The valid records use their real keys to determine the destination buckets, while the invalid records in the group use randomly assigned different keys to calculate the buckets. Now we could use the `BuildBuckets` method in Section 4.2 to conduct a round of pseudo-random communication.

Algorithm 3: Oblivious Aggregate: Stage 1

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

```
1  OSort(P, _.key) ;                                   // sort by key.
2  Aggregate P with a sequential scan; store the result in the last
     record; mark other records as invalid.
   /* Assign buckets. */
3  nonce ← 0;
4  foreach d ∈ P do
5      h ← Hash(d.key) ;                               // use real key.
6      cmov(¬d.valid, h, Hash(d.key||nonce)) ;         // use dummy key.
7      d.bucket ← h mod N;
8      nonce ← nonce + 1;
9  B ← BuildBuckets(P, N);
10 return B;
```

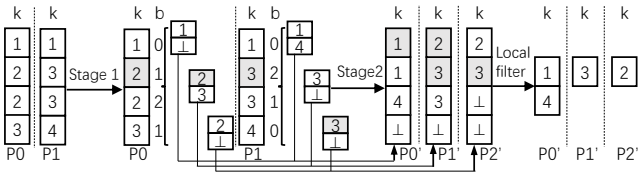


Figure 3: Overall flow of SODA oblivious aggregate. "k" and "b" stand for key and bucket. Records in dark are those turned into invalid. After stage 1, records are assigned into padded buckets according to their (pseudo-random) bucket IDs "b".

After shuffling the buckets, we continue with stage 2. Now, the records holding the locally and partially aggregated values are in the desired destinations following their keys, so they can be further aggregated through a single scan. The other dummy records have different keys from the real records and do not affect correctness.

Efficiency. Our design still has $O(n \log^2 n)$ time complexity and $O(n)$ space complexity for local processing, the same as the baseline. However, instead of a global sort, our design requires a single round of (pseudo-)random communication, plus a few times of local `OSort` and `ODistribute`. Their cost is much smaller than the baseline.

4.5 Oblivious Binary Equi-Join

As discussed in Section 3, the major difficulty to support generic binary equi-join in an oblivious way is to ensure uniform partition sizes for both the input and output data before and after the join. Similar to aggregate, join also requires the records with the same key to form a group. But rather than a single aggregated result, we now must collect all the original records from both tables in a group, which have various amounts, to one node. Our trick in implementing oblivious aggregate is thus not applicable anymore.

An intuitive solution is to manually arrange the groups into partitions to make all partitions have similar total sums of group sizes and then apply small padding to compensate for the differences. However, this is insufficient because the different join degrees in different groups still result in different output data sizes. For example, a specific partitioning could result in one group with 2 records of table A and 2 records of table B in partition 0, and another group with 1 record of table A and 3 records of table B in partition 1. Although both partitions have the same total number of 4 input records, the join results would have different sizes, $2 \times 2 = 4$ records in partition 0 and $1 \times 3 = 3$ in partition 1. Such two types of skewness are due to *input redistribution* and *join product*, respectively. They are also critical issues that cause load imbalance in the conventional insecure settings [10, 13, 14, 51, 62]. Our problem is more challenging as we can only transform data in limited oblivious ways.

Our key idea in SODA is a method named *two-level assignment*. The high-level idea is to first arrange all the *various sized* join groups into a set of *equally sized* bins (the first-level assignment). This simplifies the problem of addressing the input redistribution skew, to a task of ensuring that each partition has the same number of bins. Then, we calculate the sum of the join group products in each bin, using the value as the weight of the bin, and then organize the bins into partitions in the second-level assignment to address the join product skew. The algorithm ensures that all partitions have *the same number of bins* as well as *the same total weight* (i.e., join product) with proper padding.

Overall flow. As shown in Figure 4, we first random shuffle all the data partitions among the nodes using our random communication method in Section 4.2 (**step 1**), similar to oblivious filter (Section 4.3). The goal is to reach a sufficiently random key distribution so that we can apply the aforementioned padding bound to reduce the padding cost in later steps. Then we get the size of each group by applying oblivious aggregate as described in Section 4.4 (**step 2**). After the aggregate, each node has the size information for a non-overlapped subset of the join groups. They further communicate to derive the maximum group sizes α_A and α_B in the two tables, as well as their total numbers of records β_A and β_B. Note that the dummy records introduced in this aggregate step (shown in red in Figure 4) do not affect correctness, which are specially handled by our two-level assignment algorithm discussed below.

We then do the first-level assignment to pack groups into similarly sized bins, which are padded to exactly the same size later (**step 3**). We reduce it to bin-packing, in which the group size is used as the weight. We use practical approximate methods [15, 33], specifically the Next-Fit algorithm, to find the solution. Next, the second-level assignment further balances the join result sizes across the partitions (**step 4**). The sum of the join product sizes of all groups in a bin is used as the weight of this bin. We use an efficient method (described below) to distribute the bins to N partitions with the same number, and simultaneously ensure that the sum of join product sizes in each destination partition is bounded. Our implementation ensures obliviousness during the two-level assignment.

After finishing the two-level assignment, each group has its destination partition assigned. We next patch this assignment information back to the original tables (**step 5**). As all the original data now know the destination partitions, we can do the data shuffling according to the assignment (**step 6**). Each node then performs a local single-node join algorithm (**step 7**), e.g., [12, 34], during which the join product size in each output partition is padded to $\gamma/N + \alpha_A\alpha_B$ (see proof later), which is public information.

Now we explain the key steps 1, 3, and 4. Other steps are intuitive (e.g., step 2 directly follows Section 4.4) and thus omitted.

Step 1: random shuffle. Consider an extreme case, where all the records in a partition have the same key. Then the adversary

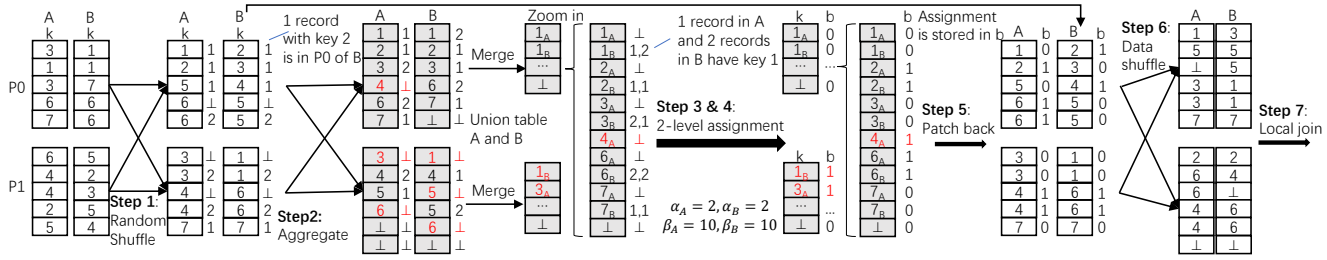


Figure 4: Overall flow of SODA oblivious join. The two-level assignment (steps 3 & 4) is further illustrated in Figures 5 and 6.

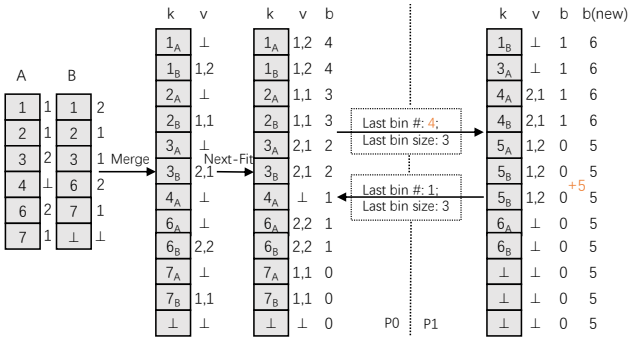


Figure 5: Oblivious join: first-level assignment. After applying aggregate to get the group sizes, we obliviously merge the two result tables and use bin-packing to pack groups into bins. "k" stands for key with subscripts denoting table A or B. "v" contains the group sizes in tables A and B. "b" denotes the bin ID assigned locally at first. After communication, the bin ID is updated to the global "b(new)".

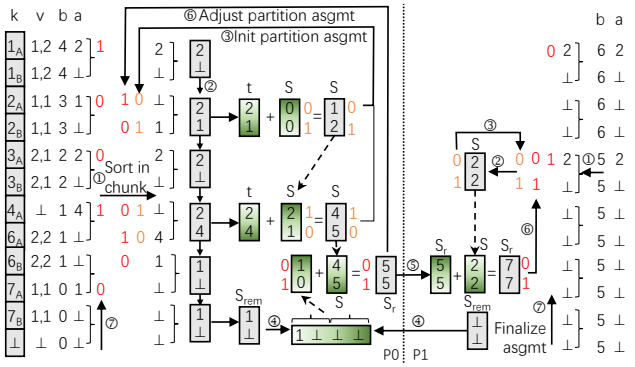


Figure 6: Oblivious join: second-level assignment. "a" stands for bin weight (sum of join product sizes). Orange denotes initial assignment; red denotes final assignment. Light green to dark green means ascending order.

can observe that all the records in this partition are going to the same destination node when we actually exchange data after the two-level assignment, which leaks information. To accommodate the worst case, each node would need to send all its records to every other node, which is too expensive. Randomly shuffling the records at the beginning reduces this padding. Even though now we introduce an additional round of communication, the total communication traffic is still smaller than the baseline, as the random communication only incurs small overheads (Section 4.2).

Step 3: first-level assignment. We use the Next-Fit bin-packing algorithm in the first-level assignment to pack groups into bins. Ohrimenko et al. [49] used bin-packing as offline pre-processing, with the First-Fit-Decreasing algorithm [15] to get a better bound. We instead aim for online processing and use Next-Fit to avoid any global oblivious sort. We set the bin capacity to a public number $c = \alpha_A + \alpha_B$ to accommodate the largest group. For $\beta_A + \beta_B$ total records, $2(\beta_A + \beta_B)/c$ bins are enough to hold all the groups for any group size distribution, as every 2 adjacent bins should have more than c records, or they could be combined. The concrete execution is as follows, and illustrated in Figure 5. We first merge the result partitions from the aggregate step of the two tables, and sort the records by (key, table) *locally* at each node, so the same keys from table A are before those from table B. Each node then does Next-Fit packing locally, with a simple scan from the bottom up, as shown in Figure 5. During the scan, the current bin size is obliviously recorded. If the current record (representing a group due to step 2 aggregate) cannot fit in the bin, a new bin is opened. When the scan finishes, the last bin in each node may still have space. All nodes then exchange their last bin information, in order to update their local bin IDs to the globally unique bin IDs (by another scan). They also possibly merge the half-full last bins during this process. In Figure 5, the last bins in the two partitions P0 and P1 cannot merge, so the bins of P1 are numbered starting from 5, after the last bin ID 4 of P0. The whole procedure is oblivious.

Step 4: second-level assignment (high-level idea). We first calculate the weight of each bin, by summing up the join product sizes of all groups in it (the leftmost part in Figure 6). Our goal is to find a partitioning scheme to ensure all the partitions have the same number of bins as well as similar weights (padded to exactly the same later). We use the following algorithm to obtain the solution. Notice that this is an abstract description that is different from our implementation described later, so the description is not oblivious and does not exclude expensive operations.

We first sort all the bins by their weights and organize the sorted bins in the row-major order into an N-column matrix denoted as M. Thus $M_{k,i} \le M_{k,j} \le M_{k+1,i}, \forall 1 \le i < j \le N$ for any row k. The matrix M forms a partitioning scheme, where the bins in the ith column are assigned to the ith partition. The column sum represents the total weight of each destination partition. We

can prove that the column sum obtained in such a simple way is bounded by $\gamma/N + \alpha_A\alpha_B$, where γ is the total weight of all bins, i.e., the output table size (excluding padding). Assume M has R rows. By accumulating the inequality above, we have $\sum_{k=1}^{R-1} M_{k,i} \le \sum_{k=1}^{R-1} M_{k,j} \le \sum_{k=2}^{R} M_{k,i}, \forall 1 \le i < j \le N$. Adding the missing $M_{R,j}$ and $M_{1,i}$ to the second and third terms yields $\sum_{k=1}^{R} M_{k,j} \le \sum_{k=1}^{R} M_{k,i} + \alpha_A\alpha_B$, because $\alpha_A\alpha_B$ is the maximum bin weight (join product size) for the maximum bin size $c = \alpha_A + \alpha_B$ from the first-level assignment. This means the difference of column sums is bounded by $\alpha_A\alpha_B$, and so the maximum is bounded by $\gamma/N + \alpha_A\alpha_B$.

Step 4: second-level assignment (oblivious and efficient implementation). Now we illustrate how we obliviously achieve the above assignment across all nodes, *without* using expensive global sort and even *without* fully sorting each local partition. At the beginning, following the results of the first-level assignment, each node now holds its packed bins with a disjoint key range from each other. Since the granularity of the second-level assignment is bins, only one record in each bin is selected as the representative. To do so, the idea is similar to that in local filter (Section 4.3). We first scan all the records on the node to select each representative and write the bin weight into it (the leftmost part in Figure 6, field "a"). The other records keep dummy weights. We omit the last bin of a node if it has been merged with other nodes during the first-level assignment. Then the representative records are split into chunks of size N_{out} (the number of output partitions), and the ith representative is obliviously placed at $i \bmod N_{\text{out}}$ in its corresponding chunk. As an example, in Figure 6, the representatives of bins 4 to 0 from top to bottom have weights of 2, 1, 2, 4, 1, respectively. They are alternately placed at positions 0 and 1 in the size-2 ($N_{\text{out}} = 2$) chunks (①).

Next, a buffer t of size N_{out} flows down the array of chunks and collects the representatives in each chunk to fill in each of the N_{out} slots (②). Then, we assign these N_{out} representatives to the N_{out} output partitions in a one-to-one manner, ensuring the same number of bins for all nodes. To further ensure similar total weights in all partitions, the assignment is based on the current weight sums in all the partitions, denoted as S. Specifically, the bin with the maximum weight in the buffer is assigned to the partition with the minimum current weight sum (③). For example, with $t = [2, 4]$ in the second buffer in the middle of Figure 6, the first bin with weight 2 is added to partition 1 with sum 2, and the other bin with weight 4 is added to partition 0 with sum 1. This updates S from $[1, 2]$ to $[4, 5]$. If the buffer is not full when it flows to the end, the remaining representatives are collected into S_{rem}.

Now we have obtained the initial assignment of bins to partitions *independently at each node*. Algorithm 4 shows the entire process so far, which only uses oblivious operations. Next, we exchange the remaining bins in S_{rem} among all nodes, merge and sort them, and determine their assignments similarly as above (④). We omit the details of this step, and only show in Figure 6 bottom where the assignment of $S_{\text{rem}} = [1, 0]$ on P0 is $[0, 1]$ (in red) for the two bins.

When finally combining the initial assignments S and S_{rem} across all nodes, we make some further cross-node adjustments to get a better bound for the total weight sum on each destination partition. Essentially, we apply the "maximum plus minimum" trick again, but now *across different nodes*. We propagate the assignment S across all nodes sequentially (⑤). Each node receives S_r from its predecessor to combine with its local S, and generates a new S_r to be sent to the successor. The first node starts with $S_r = S_{\text{rem}}$. The final S_r is the total weight sums. As shown in Figure 6, when combining $S_{\text{rem}} = [1, 0]$ with the local $S = [4, 5]$ on P0, it is better to switch the original adjustment of S from $[1, 0]$ (in orange) to $[0, 1]$ (in red) to achieve a more balanced total weight distribution of $S_r = [5, 5]$ (⑥). On P1, in contrast, there is no need to adjust the assignment. Despite being sequential, both the computation and communication are lightweight since we only process S, which only has N_{out} elements. Algorithm 5 summarizes this cross-node adjustment. It also correctly handles the assignment of the last bins using both received l_r and local l.

Algorithm 4: Oblivious Join: Second Assignment Init

Data: Merged aggregated data $\mathcal{P}$ from first-level assignment, with fields b as bin ID and a as bin weight. Weight sum array $\mathcal{S}$ and remaining weight sum array $\mathcal{S}_{\text{rem}}$, both of size N_{out}, assignment status l for last bin.

```
1  Obliviously set P[0].a to ⊥ to ignore last bin P[0], if it is merged to another node;
2  l ← UNASSIGNED ;                                   // initialize last bin status.
3  cmov(P[0].a = ⊥, l, IGNORED) ;                     // ignore if merged to another node.
4  b ← ⊥;
5  foreach d ∈ P do
6      Set new field d.idx as original sequential index;
7      c ← d.b ≠ b ∧ d.a ≠ ⊥ ;                         // condition for being representative.
8      d.wgt ← ⊥; cmov(c, d.wgt, d.a) ;               // set representative weight.
9      b ← d.b;
10 S ← {(i, 0)},   i ∈ [0, N_out) ;                   // two fields idx, wgt
11 t ← {(i, ⊥, ⊥)},   i ∈ [0, N_out) ;                // three fields idx, wgt, part
12 loc ← 0 ;                                          // in-chunk location of representative.
13 for i ← 0 to ⌈|P|/N_out⌉ − 1 do
14     ch ← P[i × N_out : (i + 1) × N_out] ;          // mutable reference of a chunk
       /* Fill in buffer t. */
15     foreach d ∈ ch do                              // set in-chunk location in buffer t
16         cmov(d.wgt ≠ ⊥, d.loc, loc mod N_out);
17         cmov(d.wgt ≠ ⊥, loc, loc + 1);
18     OSort(ch, _.loc); ODistribute(ch, _.loc);
19     cmov(ch.wgt ≠ ⊥ ∧ t.wgt = ⊥, t.wgt, ch.wgt) ; // element-wise.
       /* If buffer t is fully filled, assign to weight sum S. */
20     c ← loc ≥ N_out; loc ← loc mod N_out;          // whether t is full.
21     OSort(S, _.wgt); OSort(t, −_.wgt) ;            // in opposite order.
22     cmov(c, S.wgt, S.wgt + t.wgt) ;                // sum weights; element-wise.
23     cmov(c, t.part, S.idx) ;                       // init assignment; element-wise.
       /* Record init assignment. */
24     OSort(t, _.idx) ;                              // restore to align with ch.
25     cmov(c, ch.part, t.part) ;                     // element-wise.
       /* Chunk may have remaining bins not filled into buffer yet. */
26     for j ← 0 to N_out − 1 do
27         cmov(c ∧ j < loc, t[j].wgt, ch[j].wgt);
28         cmov(c ∧ j ≥ loc, t[j].wgt, ⊥);
       /* Record assignment for last bin if it has been assigned. */
29     cmov(c ∧ P[0].part ≠ ⊥ ∧ l = UNASSIGNED, l, P[0].part);
   /* Remaining weight sums; to be assigned. */
30 S_rem.wgt ← t.wgt;
```

Algorithm 5: Oblivious Join: Second Assignment Adjust

Input: Local and received weight sums $\mathcal{S}$ and $\mathcal{S}_r$, local and received last bin assignments l and l_r.

Output: Weight sum $\mathcal{S}_r$ and last bin assignment l_r to be sent out.

```
1  OSort(S_r, _.wgt); OSort(S, −_.wgt) ;          // in opposite order.
2  S_r.wgt ← S.wgt + S_r.wgt ;                    // element-wise.
3  S.part ← S_r.idx ;                             // adjust local assignment.
4  OSort(S, _.idx);                               // restore original order.
5  foreach s ∈ S do
6      c ← l has been assigned ∧ l = s.idx;
7      cmov(c, l, s.part) ;                       // adjust last bin assignment.
8  cmov(l = UNASSIGNED, l, S_rem[0].part) ;       // last bin in S_rem.
9  cmov(l = IGNORED, l, l_r) ;                    // last bin from other node.
10 l_r ← l ;                                      // send last bin assignment to next node.
```

We prove the bound of the "maximum plus minimum" trick. Assume $S_z = S_x + S_y$. S_x and S_y are sorted in opposite orders and both satisfy $\max S - \min S \le \alpha_A \alpha_B$. So for any i, j, $S_x[i] - S_x[j]$ and $S_y[i] - S_y[j]$ have different signs. Therefore, $|S_z[i] - S_z[j]| = |S_y[i] - S_y[j] + S_x[i] - S_x[j]| \le \max(|S_y[i] - S_y[j]|, |S_x[i] - S_x[j]|) \le \alpha_A \alpha_B$. Given that all the initial S arrays satisfy the assumption, the final S_r still satisfies the same inequality by induction. This means the difference between column sums in S_r is bounded by $\alpha_A \alpha_B$, and so the maximum is bounded by $\gamma/N + \alpha_A \alpha_B$.

The final step of the second-level assignment is to write the assignment results back to all the original representatives of the bins (⑦), including the last bin that is assigned to partition l. This follows the reverse procedure of Algorithm 4. We omit the details.

Efficiency. We emphasize that SODA oblivious join does not involve any expensive global sort. It requires 5 *data* communication rounds, namely (1) one in **step 1** random shuffle, (2) one in **step 2** aggregate, (3) one to patch the assignment back in **step 5**, (4) one to shuffle data for the single-node join in **step 6**, and (5) one at the end to filter dummy records. Note that (2) and (3) do not involve the entire rows of the original data tables. In contrast, the baseline join (Figure 1) requires 9 communication rounds, including 8 from 2 column sorts (one for join and the other for filter) and 1 round for the maximum group communication. The asymptotic complexity of local processing remains the same as the baseline, i.e., $O(n \log^2 n)$ time and $O(n)$ space.

For communication padding, the bound on the number of rows in each bucket produced by each input partition for (4) is $\mu + k\sqrt{\mu}$ following Section 4.2, where $\mu = 2|\mathcal{P}|/N = 2(\beta_A + \beta_B)/N^2$. The factor of 2 is from the first-level assignment. We could use the (pseudo-)random communication bound in Section 4.2 because of the random shuffle at the beginning. Different from communication padding, join result padding is specific to join, greatly affecting the performance. SODA reduces the final join result padding from $\min\left(\left\lfloor \frac{n}{\alpha_A+\alpha_B} \right\rfloor \cdot (\alpha_A \alpha_B) + \left(\frac{n \bmod (\alpha_A+\alpha_B)}{2}\right)^2, \gamma\right)$ to $\gamma/N + \alpha_A \alpha_B$, reducing the data volume in local sort during the single-node join.

5 SECURITY ANALYSIS

We prove the security of our designs by showing the existence of a simulator Sim defined in Section 2.4, which only has access to the six public parameters. Since an adversary cannot distinguish between the real trace and the simulated trace, it cannot learn extra information from a real execution.

For local processing, we enforce the following properties in both the high-level algorithms and our implementations (see details in [40]): (1) all loop conditions depend only on public information irrelevant to sensitive data; (2) all branches are converted into `cmov` instructions to be oblivious to branch conditions; (3) all memory/disk accesses either follow a sequential scan manner, or use oblivious primitives `OSort`, `OMerge`, and `ODistribute`, which themselves are oblivious according to Section 4.1. Therefore, the execution traces are independent of the sensitive data content. For example, the trace of a scan on some intermediate data (e.g., buckets, chunks) contains sequential addresses with the length decided by the data size. Since Sim knows the data size, it can generate an indistinguishable trace.

For network communication, as long as the communicated data volume between each pair of source and destination nodes is independent of the sensitive data, Sim can transfer the same amount of data locally computed from random input (indistinguishable if encrypted) between the nodes. This condition holds in SODA. For each input partition, its outgoing data to each output partition are either perfectly balanced or padded by our (pseudo-)random communication primitive in Section 4.2. In both cases, the size is fully determined by the input partition volume, which is initially a public value (Section 2.4). More specifically, for global filter and aggregate, there is only one round of data communication padded by (pseudo-)random communication. For join, we first do a random shuffle and an aggregate, both being oblivious. During the two-level assignment, we send metadata among all partitions with a fixed number of rounds. All metadata are in the form of vectors of a fixed length N_{out}. The next step sends buckets back to the senders to patch the assignments to the original tables. This data communication is the inverse of the aggregate operation and is also oblivious. In the final data shuffling, an input partition sends the same amount of records to each output partition, after padded similarly to our (pseudo-)random communication. Thus, communication during the whole procedure is independent of sensitive data.

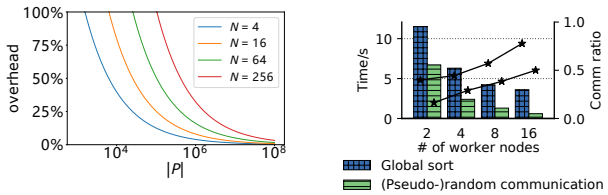
6 EVALUATION

6.1 Implementation

Both the baseline in Section 3 and SODA are implemented in Rust on top of Flare [39], an open-source, Rust-based, Spark-like secure analytics framework with competitive performance and much simpler code structures, making performance and security analysis easier. We use Intel SGX as our TEE choice, with Rust SGX SDK v1.1.3 [17, 63] for implementing the code in the trusted domain. We remark that our proposed algorithms are not tied to Spark or SGX. They are applicable to other frameworks and TEEs as well. All the local processing stages in the algorithms described in Section 4 execute in the trusted domain, and are doubly oblivious [46].

6.2 Experimental Setup

Platform. We use multiple cloud instances of type `ecs.g7t.4xlarge` on the Alibaba Cloud, which are equipped with Scalable SGX possessors with GB-level trusted memory. The inter-machine network bandwidth is up to 25 Gbps.



(a) Theoretical padding overhead. **(b) Performance.**

Figure 7: Analysis of (pseudo-)random communication.

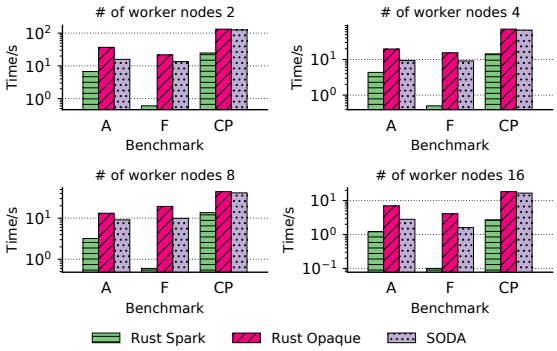


Figure 8: Performance on micro-benchmark set (1).

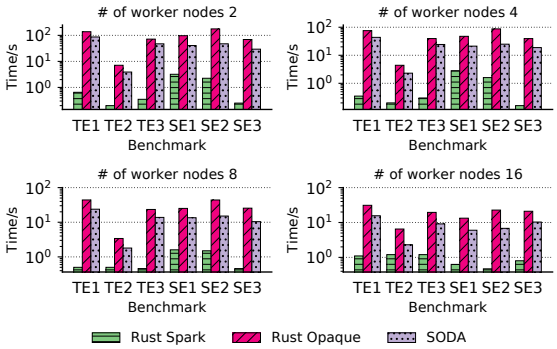


Figure 9: Performance on micro-benchmark set (2).

Workloads. We use both micro- and macro-benchmarks. The micro-benchmarks include two suites. (1) Big Data Bench [64], similar to AMP Big Data Benchmark [4], which was used by Opaque [70]. We run Aggregation (A), Filter (F), and Cross-Project (CP) queries, similarly as in Opaque. Here CP is actually a primary-foreign join. (2) Queries TE1-TE3 and SE1-SE3 used in [12], on the TPC-H dataset and the social graph dataset [11]. They are mainly used for evaluating binary equi-join with various join degrees. Same as in [12], we generate three tables "popular-user", "inactive-user" and "normal-user", and randomly sample 1 to 100 MB as our dataset.

While the micro-benchmarks only evaluate a single operator, the macro-benchmarks involve more oblivious operators, which are more complex. We construct them from a diverse set of benchmarks in the domains of data analytics, graph processing, and machine learning. Specifically, we include PageRank (PR), Transitive Closure (TC1), Dijkstra (Dij), Triangle Counting (TCo), K-Means (KM), and Matrix Multiplication (MM). They are widely used in many frameworks [18, 39, 54, 58, 70]. We run them on real-world datasets [37, 38, 43]. For all the queries, we manually apply commonly used query optimizations (e.g., selection pushdown) and generate optimized physical plans, due to current lack of support in Flare [39] and its insecure baseline Vega [56].

Baselines. We compare three systems. (1) Rust Spark, the original Vega [56] as the insecure reference. (2) Rust Opaque, the baseline in Section 3. (3) SODA, our proposal. Notice that we could not directly compare with the original Opaque framework [70] because its oblivious mode is not open sourced. For all benchmarks, we evaluate four system configurations with 2, 4, 8, and 16 nodes.

6.3 Performance on Micro-Benchmarks

We first compare (pseudo-)random communication against global sort. Figure 7a shows the bucket padding overhead compared to the original bucket size for various data partition sizes $|\mathcal{P}|$ and numbers of nodes N, under the failure rate of 2×10^{-57}. The overhead is only moderate in typical setups, e.g., less than 8% when using $N = 16$ to process 1 million records, or less than 32% even when $N = 256$. In contrast, global sort is equivalent to having a 300% overhead due to four communication rounds. Figure 7b shows the total execution time of the two primitives and the portions of communication time, when processing 1 GB data. With more nodes in the cluster, the communication time becomes increasingly dominant. The speedup of our primitive is significant, ranging from 2× to 6×.

Next, we evaluate micro-benchmark set (1). The results are shown in Figure 8. For A, F, and CP, their speedups averaged over the four different system scales are 2.6×, 1.8× and 1.1×, respectively. The speedups of SODA oblivious aggregate and filter are higher, because they both require small amounts of communication and local sorts. For CP, it is expected that the performance of the baseline join approaches that in SODA, since CP performs the primary-foreign join, in which case the baseline would not generate much join result padding. On the other hand, the communication volume of SODA is relatively large due to the padding to the bound in the Next-Fit algorithm, which is around 2× expansion. Nevertheless, SODA join is still slightly better due to fewer communication rounds.

On the other hand, when compared to the insecure system, the security overheads of SODA for these three workloads are 2.4×, 15.0×, and 6.2×, respectively. These numbers are only moderately large, except for F (filter). This is because insecure filter does not need cross-node data shuffle at all, while oblivious filter additionally introduces a round of communication. For aggregate and join, both the insecure and oblivious versions need global data exchange.

Then we evaluate micro-benchmark set (2), where all the join queries are general binary equi-join. As shown in Figure 9, for TE1-TE3, the speedups range from 1.6× to 2.0×, 1.8× to 2.8×, and 1.5× to 2.2× as the number of nodes increases from 2 to 16. For SE1-SE3, the speedups range from 1.8× to 2.2×, 2.9× to 3.7×, and 2.0× to 2.4×. These performance improvements are more significant than the CP workload because the baseline cannot efficiently handle general binary equi-join, as discussed in Section 3. As the cluster size grows,

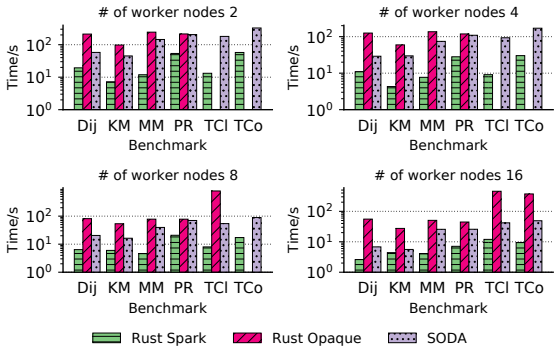


Figure 10: Performance on the macro-benchmark set. The missing bars are due to out-of-memory crashes.

Table 1: The relationship between padding reduction and speedup in micro- and macro-benchmarks.

Micro-Benchmarks	CP	TE1	TE2	TE3	SE1	SE2	SE3
Padding reduction	9.5	1.1	1.2	1.2	3.9	11.4	1.7
Speedup	1.2	2.0	2.8	2.1	2.2	3.4	2.0

Macro-Benchmarks	MM	PR	TCl	TCo
Padding reduction	0.9	1.1	9.6	7.4
Speedup	2.0	1.7	10.9	7.6

the speedup slightly increases. The baseline sequentially passes significant data from one node to the other (Figure 1), hindering scalability, while SODA communicates only small metadata.

When compared to the insecure system, SODA still incurs large overheads for join, due to several costly local sort primitives, which are not needed in the insecure version. Fortunately, the sort primitive could largely benefit from distributed execution, and the slowdown of SODA over the insecure system reduces from 47.2× to 5.5× (in geometric mean) as the number of nodes increases from 2 to 16. This result demonstrates the good scalability of our proposed oblivious join algorithm in SODA.

6.4 Performance on Macro-Benchmarks

Finally, we evaluate the macro-benchmarks that are consisted of multiple oblivious operators. As shown in Figure 10, the performance gains range from 1.1× to 14.6×. PR has the lowest speedup since its join type is primary-foreign join, and the join dominates the execution. SODA brings significant benefits on graph algorithms, such as Dij with an average 4.7× speedup, TCl with 10.9×, and TCo with 7.6×. These algorithms involve more padding in the baseline due to join on real-world graph datasets (see Section 6.5). Notice that some workloads even crash on the baseline framework because the excessively large amount of data padding exhausts memory.

6.5 Padding Reduction Analysis

We further illustrate the correlation between the padding reduction of join results and the speedup in binary equi-join. The padding reduction is the ratio of padded join result volumes between the baseline and SODA. Here we exclude the workloads that do not involve join. As shown in Table 1, for most workloads, the speedup is well correlated with the padding reduction. For MM, despite negative padding reduction (< 1), SODA still achieves speedup thanks to communication volume reduction and less computation. However, CP and SE2 show significant padding reduction but small performance gains. To find out the reason, we enlarge the datasets by 5×. The speedup in SE2 now grows to 18×, better matching the padding reduction. However, CP still exhibits a low speedup. We find that although the dataset is GB-level large, the join result of CP is quite small, with only hundreds of output records. Padding on such a small output does not affect the overall performance much.

7 RELATED WORK

Oblivious algorithms for data analytics have been widely studied recently [6, 12, 18, 19, 34, 41, 46, 49, 66, 70]. For single-node query processing, most work focused on join, the most challenging task. Li et al. [41] were the first to construct oblivious join with secure processors, including binary equi-join, band join, and acyclic multi-way equi-join. Arasu et al. [6] also proposed other oblivious operators beyond join, but the details were missing [34]. OBliDB [19] supported selection (a.k.a., filter), aggregate, and binary equi-join. Krastnikov et al. [34] concentrated on building a more efficient binary-equi join. The most recent state-of-the-art work [12] was the first to integrate ORAM [52, 60, 65] to build oblivious join.

On the other hand, for distributed query processing, the leakage would be more severe due to the additional network traffic side channel. Ohrimenko et al. [49] defined two levels of obliviousness and proposed corresponding algorithms under the MapReduce paradigm [16]. Another work M2R [18] also focused on MapReduce, but only hided correlation information, which was the weaker level in [49]. ObliDC [66] focused on security formalization rather than performance. Opaque [70] was the state-of-the-art work based on Spark. It utilized column sort to construct oblivious algorithms. However, it did not support binary equi-join, and column sort is performance-wise expensive.

8 CONCLUSIONS

We propose SODA, a set of efficient algorithms for oblivious data analytics operators in distributed processing scenarios, including filter, aggregate, and equi-join. Compared to previous systems, SODA completely eliminates the use of expensive global sort primitives, and minimizes the necessary data padding, therefore achieving significant performance improvements of 1.1× to 14.6× on complex data analytics benchmarks. In addition, SODA also extends the functionality, as its equi-join operator generalizes beyond the primary-foreign join in the baseline system.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262) and the Institute for Interdisciplinary Information Core Technology, Xi'an. Mingyu Gao is the corresponding author.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An O(n log n) Sorting Network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*. 1–9.

[3] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security. *White paper* (2004).

[4] AMPlab, University of California, Berkeley. 2014. *Big Data Benchmark*. Retrieved Oct 30, 2022 from https://amplab.cs.berkeley.edu/benchmark/

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[6] Arvind Arasu and Raghav Kaushik. 2014. Oblivious Query Processing. In *Proceedings of the 17th International Conference on Database Theory (ICDT)*. 26–37.

[7] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1383–1394.

[8] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *Symposium on Simplicity in Algorithms (SOSA)*. 8–14.

[9] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*. 207–218.

[10] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. 2014. Advanced Join Strategies for Large-Scale Distributed Computation. *Proc. VLDB Endow.* 7, 13 (2014), 1484–1495.

[11] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*.

[12] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[13] Qi Chen, Jinyu Yao, and Zhen Xiao. 2014. Libra: Lightweight Data Skew Mitigation in MapReduce. *IEEE Transactions on Parallel and Distributed Systems* 26, 9 (2014), 2520–2533.

[14] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 63–78.

[15] Edward G Coffman, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. Bin Packing Approximation Algorithms: Survey and Classification. In *Handbook of Combinatorial Optimization*. 455–531.

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[17] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. 2017. Poster: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2491–2493.

[18] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in MapReduce Computation. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. 447–462.

[19] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.

[20] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 275–294.

[21] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015. Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM. *IACR Cryptol. ePrint Arch.* (2015).

[22] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES)*. 251–261.

[23] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. 182–194.

[24] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[25] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 599–613.

[26] Michael T Goodrich. 2014. Zig-Zag Sort: A Simple Deterministic Data-Oblivious Sorting Algorithm Running in O(n log n) Time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. 684–693.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[28] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.* 2016 (2016), 204.

[29] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[30] Intel. 2017. *Memory Encryption Technologies Specification*. Retrieved June 30, 2022 from https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf

[31] Intel. 2018. *Intel Software Guard Extensions (Intel SGX) Developer Guide*. Retrieved June 30, 2022 from https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html

[32] Intel. 2020. *Intel TDX*. Retrieved September 30, 2022 from https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[33] David S Johnson. 1973. *Near-Optimal Bin Packing Algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[34] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 12 (2020), 2132–2145.

[35] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. 557–574.

[36] Tom Leighton. 1985. Tight Bounds on the Complexity of Parallel Sorting. *IEEE Trans. Comput.* 100, 4 (1985), 344–354.

[37] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD)*. 177–187.

[38] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[39] Xiang Li, Fabing Li, and Mingyu Gao. 2023. Flare: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. *Proc. VLDB Endow.* 16, 6 (2023), 1439–1452.

[40] Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. 2022. *SODA code repository*. https://github.com/tsinghua-ideal/flare/tree/oblivious_soda

[41] Yaping Li and Minghua Chen. 2008. Privacy Preserving Joins. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE)*. 1352–1354.

[42] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-Core Platforms. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. 865–880.

[43] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NeurIPS)*, Vol. 1. 539–547.

[44] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[45] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[46] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*. 279–296.

[47] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press.

[48] David Nassimi and Sartaj Sahni. 1979. Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Trans. Comput.* 28, 1 (1979), 2–7.

[49] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. 2015. Observing and Preventing Leakage in MapReduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1570–1581.

[50] Rafail Ostrovsky. 1990. Efficient Computation on Oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*. 514–523.

[51] Orestis Polychroniou, Wangda Zhang, and Kenneth A Ross. 2018. Distributed Joins and Data Placement for Minimal Network Traffic. *ACM Transactions on Database Systems (TODS)* 43, 3 (2018), 1–45.

[52] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2014 (2014), 997.

[53] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*.

[54] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. 38–54.

[55] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 3–24.

[56] Raja Sekar et al. 2019. *Vega*. Retrieved June 30, 2022 from https://github.com/rajasekarv/vega

[57] AMD SEV-SNP. 2020. Strengthening VM Isolation with Integrity Protection and More. *White Paper, January* (2020).

[58] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework with Trusted Processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1211–1228.

[59] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[60] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[61] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. 1041–1056.

[62] Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. 2016. Load Balancing and Skew Resilience for Parallel Joins. In *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 313–324.

[63] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2333–2350.

[64] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. 2014. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[65] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 850–861.

[66] Pengfei Wu, Qingni Shen, Robert H Deng, Ximeng Liu, Yinghui Zhang, and Zhonghai Wu. 2019. ObliDC: An SGX-Based Oblivious Distributed Computing Framework with Formal Proof. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS)*. 86–99.

[67] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. 640–656.

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–28.

[69] Mark Zhao and G. Edward Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy (S&P)*. 229–244.

[70] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 283–298.