# JODES: Efficient Oblivious Join in the Distributed Setting


Yilei Wang, Xiangdong Zeng, Sheng Wang, Feifei Li

{fengmi.wyl,zengxiangdong.zxd,sh.wang,lifeifei}@alibaba-inc.com

Alibaba Cloud

Hangzhou, China



## ABSTRACT

Trusted execution environment (TEE) has provided an isolated and secure environment for building cloud-based analytic systems, but it still suffers from access pattern leakages caused by side-channel attacks. To better secure the data, computation inside TEE enclave should be made oblivious, which introduces significant overhead and severely slows down the computation. A natural way to speed up is to build the analytic system with multiple servers in the distributed setting. However, this setting raises a new security concern—the volumes of the transmissions among these servers can leak sensitive information to a network adversary. Existing works have designed specialized algorithms to address this concern, but their supports for equi-join, one of the most important but non-trivial database operators, are either inefficient, limited, or under a weak security assumption.

In this paper, we present JODES, an efficient oblivious join algorithm in the distributed setting. JODES prevents the leakage on both the network and enclave sides, supports a general equi-join operation, and provides a high security level protection that only publicizes the input sizes and the output size. Meanwhile, it achieves both communication cost and computation cost asymptotically superior to existing algorithms. To demonstrate the practicality of JODES, we conduct experiments in the distributed setting comprising 16 servers. Empirical results show that JODES achieves up to a sixfold performance improvement over state-of-the-art join algorithms.



**PVLDB Reference Format:**
Yilei Wang, Xiangdong Zeng, Sheng Wang, Feifei Li. JODES: Efficient Oblivious Join in the Distributed Setting. PVLDB, 18(5): 1291 - 1304, 2025.
doi:10.14778/3718057.3718060

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/Aqua-Dream/Jodes/.


## 1 INTRODUCTION

The provision of computation over encrypted data has become a crucial offering for cloud-based analytic system providers [4, 35, 52, 60]. The importance of encryption lies in its ability to ensure the confidentiality and privacy of data processed in the cloud, especially given the large amounts of sensitive and confidential information from enterprise customers. Data such as personal details, financial documents, trade secrets, and intellectual property require strong security measures to prevent unauthorized access or breaches. The deployment of encrypted data analytic systems [4, 5, 10, 18, 24, 36, 44, 45, 50, 52, 59] guarantees the secure computation and transmission of data, ensuring that only authorized parties can access and decipher the information. This capability grants customers more authority over their data, enabling them to adhere to data governance protocols and address issues revolving around data privacy and sovereignty.

However, computation over encrypted data often incurs significant overhead, typically resulting in several orders of magnitude slowdown. For instance, solutions based on secure multi-party computation [10, 43] or homomorphic encryption [46] can introduce at least three orders of magnitude slowdown in computations. Meanwhile, the more practical approach builds encrypted analytic systems based on trusted execution environments (TEEs) like Intel SGX [4, 18, 24, 36, 50, 52, 59], which is the focus of this paper. In this setup, cloud services operate within isolated *enclaves* where confidential data is processed. Data must be decrypted only within the enclave for computation and re-encrypted before leaving the enclave, resulting in unavoidable encryption/decryption overhead. Besides, TEEs still suffer from an important vulnerability known as the *access pattern leakage* caused by side-channel attacks [41, 56] where the host system can infer auxiliary information of encrypted data by monitoring memory accesses of the application [30, 61]. To mitigate this issue, computation in enclave should be designed as *oblivious* [14, 32], ensuring that the access pattern of the computation is independent of the input. General techniques such as Oblivious RAM (ORAM) [21] can be employed to transform non-oblivious algorithms into oblivious counterparts. However, even the most practical ORAM scheme [49] could significantly increase the running time by a factor of $O(\log^2 N)$ where $N$ is the input size, and the slowdown factor is between 90 and 450 according to the experiments of database join in [14]. Such a high overhead prevents the encrypted analytic system from practicality.

There are two directions to speed up the oblivious computation: (1) Design a specialized oblivious algorithm for each operator, which typically reduce the $O(\log^2 N)$ blow-up factor of ORAM to $O(\log N)$, *e.g.*, sorting [9] and some database operations [6, 32]; (2) Leverage the *distributed setting* [15, 51, 58], which distributes intensive computation tasks to multiple servers, therefore offsetting the overhead brought by obliviousness. Above two directions are orthogonal, and this paper studies their intersection point—designing efficient specialized oblivious algorithms in the distributed setting.

As firstly pointed out in [41], building encrypted analytic systems in the distributed setting raises new security concerns, even though data are encrypted and processed inside enclave with obliviousness protection. Consider a network adversary that observes the communications between the servers. Although data are encrypted,




Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718060

their volumes are revealed, which introduce the *communication pattern leakage*, *i.e.*, the number of elements transmitted between each pair of servers leaks information. For example, a hash join operator will gather all the tuples that have the same join keys to the same server, which is determined by the hashing values. The network adversary could therefore infer some information about the distribution of join keys by analyzing the communication pattern. To clearly distinguish the two types of leakages, we follow [13] and define a distributed algorithm to be *communication oblivious*, if its communication pattern is independent of the input (see Section 2.3 for a formal definition). The communication obliviousness defends against the network adversary. Accordingly, *computation oblivious* is defined over a local computation, meaning the memory access pattern of the computation inside enclave is independent of the input, which defends against a memory adversary. All algorithms proposed in this paper are both communication and computation oblivious, which we simply refer to as *oblivious*. For the case that only communication obliviousness is required, *e.g.*, the servers are fully trusted, the system could implement the local computations in a natural way without the computation obliviousness requirement, which we ignore the details.

## 1.1 Previous Work

Opaque [59] and SODA [37] are the only two prior works that present specialized oblivious distributed algorithms for join. Opaque starts by proposing the oblivious sorting algorithm based on column sort [33], and leverages it to implement oblivious filter, aggregate, and join. Opaque's join algorithm, however, is limited to only *primary key join* (PK join), a special type of join where tuples from one of the input tables have unique join keys. SODA [37] considers column sort to be too expensive, so it proposes its own oblivious algorithms for filter, aggregate, and join without relying on oblivious sorting. SODA's join supports a general binary equi-join, not limited to any special type of join. Nevertheless, it requires publicizing the *degrees* (*i.e.*, frequencies) of the most popular keys of the two tables, hence has a lower security level compared to other algorithms. In SODA's join algorithm, publicizing the maximum degrees is necessary for grouping tuples with the same keys to the same *bins* to perform join. To achieve obliviousness, all bins should be padded to the same volume, which is computed from the maximum degrees. We note that similar challenge also exists in the *standalone setting* (*i.e.*, a single machine is utilized to perform the join), until the oblivious *expansion* algorithm appears [32]. It then becomes the core building block of state-of-the-art oblivious standalone join algorithm [32] without requiring any public degree information of the input tables. In this paper, we propose the first distributed expansion algorithm which also serves as one of the basic primitive of our join algorithm Jodes.

## 1.2 Our Contribution

The major contribution of this paper is Jodes (Algorithm 4), an efficient Join algorithm that is Oblivious in the DistributEd Setting. Compared with previous works, it has the following advantages:

(1) Unlike Opaque's join, Jodes supports a general binary equi-join operation, not limited to a primary key join. It also does not publicize any degree information as SODA's join, thus achieving higher security level than it (Section 2).
(2) Jodes is the first oblivious distributed join algorithm that achieves communication cost linear to only input size and output size. It also has computation cost asymptotically better than all existing works (Table 2).
(3) Our experiments demonstrate that Jodes outperforms all baselines. For example, it can finish a join that outputs $1.9\times 10^8$ rows in 86s using 16 servers, which is only 1/6 of time taken by the the state-of-the-art distributed or standalone join algorithm (Section 6).

Apart from the expansion primitive, Jodes also takes shuffle, sorting, and PK join as primitives. For sorting, we simply adopt the column sort in Opaque, while for shuffle and PK join, we have dedicatedly design faster oblivious algorithms for them (Section 3). Experiments show that our algorithms for these operators improve the baselines by at least 60% (Section 6). These operators are basic and instrumental, and our improvements should be of independent interest to the community of oblivious query processing beyond the scope of Jodes itself. Any future refinement to the sorting operator can also enhance the performance of Jodes.

**Table 1: Notations used in the paper**

| Notation | Meaning |
|---|---|
| $p$ | Num of servers |
| $[p]$ | The set $\{1, 2, \ldots, p\}$ |
| $N$ (*resp.* $M$) | Num of input (*resp.* output) elements/tuples |
| $n$ (*resp.* $m$) | $N/p$ (*resp.* $M/p$) |
| $\sigma$ | Security parameter |
| $X[i]$ | The part of elements/tuples in $X$ that are located on the $i$-th server |
| $n_i$ | $\lvert X[i]\rvert$, num of elements/tuples on the $i$-th server |
| $Y_{ij}$ | The subset of elements in $X_i$ that will be sent to the $j$-th server through communication |
| $U_i$ | The target size for padding of $Y_{ij}$ for any $j \in [p]$ |

# 2 PRELIMINARY

The frequently used notations are summarized in Table 1.

## 2.1 Distributed Setting

In the *distributed setting*, there are $p$ servers that work collaboratively to execute a computational task as dictated by a specific algorithm, which we refer to as a *distributed algorithm*. Each server holds a portion of the input, which consists of $N$ elements almost equally distributed among them; the $i$-th server possesses a subset with $n_i = \Theta(N/p)$ elements, constituting its initial local dataset. The values $N$, $p$, and $\{n_i\}_{i=1}^{p}$ are public to all servers. The servers complete the task over several *rounds*. In each round, the $i$-th server processes its local dataset $X_i$ and generates output in the form $Y_{i1}, \ldots, Y_{ip}$, for $i \in [p]$. This marks the computation phase for this round. Following this, the servers enter the communication phase where all pairs of servers exchange data over a complete network. Specifically, the $i$-th server transmits the dataset $Y_{ij}$ to the $j$-th server. For any given server, the collective data received from all other servers are amalgamated to form its updated local dataset. This updated dataset is then either utilized in the subsequent round of computation, integrated into the input for the succeeding task, or emitted as the output.

To differentiate from the distributed setting, we employ the term *standalone setting* to refer to the standard setting when there is only one server. An algorithm that is designed for the standalone setting is called a *standalone algorithm*, which involves local computation on the server without any network communication.

## 2.2 Encrypted Analytic System

We focus on the cloud-based encrypted analytic system. The data on the servers is uploaded by one or more data owners in the encrypted form. When a client (who could also be one of the data owners) submits an (authorized) query to the servers, they translate the query to a query plan, *i.e.*, a series of database operations, and then execute the operations following the underlying algorithms. The output of the last operation, which is also the output of the query, is then sent to the client, also in the encrypted form. The client has the key and can decrypt the query result into plaintext. This system securely enables querying across data from different owners, which may distrust each other as well as the servers.

In the system, each server is equipped with TEE, the size of whose protected memory is large enough to hold elements located in this server during computation. This is a reasonable assumption, thanks to the second-generation Intel SGX that supports enclave size up to 128GB or even larger [17]. All data elements are safeguarded with encryption while residing outside of the enclave, *i.e.*, elements are in plaintext form inside enclave and are in ciphertext form outside enclave. The encryption scheme should defend against chosen-plaintext attack so that ciphertexts consistently appear random to the adversary, regardless of whether the corresponding plaintexts are identical. The length of a ciphertext is linear to the length of its underlying plaintext, which is independent of the plaintext's actual value. For example, AES with GCM encryption mode is sufficient. The TEEs should hold all DEKs (data encryption keys) of the data owners and the client, so that the input and output can be correctly decrypted and encrypted respectively. A common DEK for intermediate computations is held by all the TEEs on the servers, allowing ciphertexts received from one server to be correctly decrypted in enclave of another server. In the descriptions of our algorithms, we refer to an element without explicitly distinguishing its form, as it is clear that a plaintext (*resp.* ciphertext) is always inside (*resp.* outside) enclave.

## 2.3 Security Definition

*Communication oblivious.* In the distributed setting, for any (deterministic or randomized) algorithm $\mathcal{A}$ and any input $X$, let $S(X) = \{s_{ijk}\}$ be a sequence where $s_{ijk}$ is the size of messages that the $i$-th server sends to the $j$-th server in the $k$-th round. Note that if $\mathcal{A}$ is random, then each $s_{ijk}$ is a random variable. This sequence $S(X)$ is all information that a network adversary can observe, which we call the *transcript* of $\mathcal{A}$. We then have the following definition:

*Definition 2.1.* $\mathcal{A}$ is communication oblivious if there exists a probabilistic simulator Sim such that, for any input $X = (X_1, \ldots, X_p)$ where $|X_i| = n_i$ for all $i \in [p]$, the simulator can generate the simulation $\bar{S} = \mathrm{Sim}(n_1, \ldots, n_p)$ such that, there is no polynomial-time algorithm that can distinguish between the transcript $S(X)$ and the simulation $\bar{S}$ with success probability more than $1/2$.

In other words, the transcript of any input is only dependent on the input sizes across the servers, which ensures the network adversary infers no information of the input from the transcript (despite the input sizes). Note that the input sizes are usually assumed to be public. One may need to further protect them by, for example, differential privacy [16], which is out the scope of this paper.

*Computation oblivious.* For any standalone algorithm $\mathcal{A}$ with input $X$ performed *locally* on any server, its memory access operations of computing $\mathcal{A}(X)$ can be expressed by the sequence $((\mathrm{op}_1, a_1, x_1), \ldots, (\mathrm{op}_k, a_k, x_k))$, where each $\mathrm{op}_i$ represents either a read or a write operation. Specifically, a read operation retrieves the element located at the $a_i$-th position inside enclave memory, while a write operation updates the element at the $a_i$-th position to $x_i$. The *memory access pattern* of $\mathcal{A}$ with input $X$ is defined as $A(X) \coloneqq (a_1, \ldots, a_k)$, which is all the information that a memory adversary can observe through TEE side-channel attacks. We define computation oblivious as below:

*Definition 2.2.* $\mathcal{A}$ is computation oblivious if there exists a probabilistic simulator Sim such that, for any input $X$ with $|X| = n$, the simulator generates $\bar{A} = \mathrm{Sim}(n)$ such that no polynomial-time algorithm can distinguish between $A(X)$ and $\bar{A}$ with success probability more than $1/2$.

*Security parameter.* Our algorithms may fail to return correct answer. We define $\sigma$ as the *security parameter*, and the theoretical analysis ensures that the failure probabilities of our algorithms are all bounded by $2^{-\sigma}$. Note that the adversary could not observe any failure, but the potential reaction to the failure may leak information, *e.g.*, the client may re-submit the same query when a wrong answer is detected. Therefore, it is necessary to choose a sufficiently large $\sigma$ (say, $\sigma = 40$) so that the failure becomes almost impossible.

*Dummy.* During the execution of an oblivious algorithm, both computation and communication may involve *dummy* elements. These elements serve as placeholders to maintain the algorithm's obliviousness without realistic meaning, as opposed to *real* elements. For example, to achieve communication oblivious, the $i$-th server may pad messages with dummy elements to a public and data independent size $U_i$ before sending to the $j$-th server (Section 3.2). To implement dummy elements, one may assign a unique value to it, guaranteed not to occur within the actual data domain, or alternatively, append an additional attribute to each element, indicating it as either dummy or real. Regardless of the chosen implementation strategy, it is crucial that dummy elements remain imperceptible to the adversary that: (1) the ciphertexts of dummy and real elements are indistinguishable, and (2) the access patterns to dummy and real elements are indistinguishable.

## 2.4 Cost Model

In evaluating a distributed algorithm, we consider both the *communication cost* and the *computation cost*. Regarding theoretical analysis, we treat each element (or tuple, in cases where the input comprises sets of tables) as a basic unit, and define input size (*resp.* output size) as the total number of elements or tuples within the input (*resp.* output). The communication cost is the total number of elements that are communicated across the servers during the execution of an algorithm. Algorithms in Opaque [59], SODA [37], and this paper, have communication cost linear to the input and output size, so we do not hide the constant of the linear term for detail comparison. However, we do neglect the lower-order terms. For instance, the communication cost of our oblivious shuffle by key algorithm is $N + o(N)$, and we omit the $o(N)$ term in our discussions. Meanwhile, the computation cost of an algorithm is the total costs of all the rounds, where the cost of each round is defined as the maximum cost of local computations of all the servers in the round. Computation cost typically scales superlinearly, especially with computation obliviousness. We express computation costs using asymptotic notations.

In the encrypted model, data are encrypted prior to communication and re-decrypted afterward. We incorporate these encryption and decryption costs into the communication cost since they are performed (and only performed) before and after communication, and their costs are also linear to the number of elements exchanged between servers. Specifically, the time incurred by communication costs is proportional to a combination of network bandwidth and the speeds of decryption and encryption, introducing a considerable constant factor to the overall runtime. In contrast, computation cost, while superlinear, typically has a smaller constant factor dependent on CPU clock speed, memory frequency and latency, *etc.* Given the varying significance of each cost in different scenarios, we strive in this paper to minimize both to the greatest extent possible.

*Parameter assumptions.* Commonly, the security $\sigma$ is set between 40 and 80 in the literature [19, 53]. Given this context, our study will focus on inputs with size $N$ that satisfies both $N = \omega(p^2\sigma)$ and $N = O(2^\sigma)$. This ensures that $N$ is not overly small (rendering the distributed setting superfluous) nor excessively large (exceeding contemporary servers' processing capabilities). In the join algorithm, the costs are also determined by the public output size $M$. We make similar assumptions to $M$, *i.e.*, $M = \omega(p^2\sigma)$ and $M = O(2^\sigma)$. Note that if $M$ is too small, the number of servers $p$ may be correspondingly reduced (in either a logical or physical sense) subsequent to the join operation, or it might even revert to the standalone setting. We leverage these assumptions to simplify complexity expressions throughout this paper, *e.g.*, it is obvious that $\log N = \Theta(\log n)$ and $\log m = \Theta(\log M) = O(\log N)$, where $n = N/p$ and $m = M/p$. Note that these assumptions only affect the performance analysis; correctness and security of our algorithms remain intact.

## 2.5 Output Padding

The security definition in Section 2.3 assumes the input size $N$ is public but not the output size $M$ as it is data dependent. However, the worst-case output size of join operations can grow exponentially, resulting in poor performance or even system unavailability. Consequently, researchers working on oblivious query processing have chosen to sacrifice certain security measures to achieve better performance. Specifically, they have proposed various padding schemes to standardize the output size of each join operation. Detailed explanations can be found in the full version of the paper [55].

# 3 BASELINE

In this section, we review state-of-the-art distributed algorithms for the operators utilized by our join algorithm, as well as the sole existing oblivious distributed join algorithm. Our enhancements to some of these algorithms are presented in Section 4.

## 3.1 Computation Oblivious Primitives

Below we introduce some oblivious primitives that will be used in the local computations of our algorithms. All primitives introduced in this section only run *locally* in the standalone setting, *i.e.*, they are standalone primitives, therefore in Section 3.1 by "oblivious" we mean computation oblivious. These primitives form the basic building blocks to achieve computational obliviousness.

We denote the oblivious computation of the assertion $c$ as a binary value with $[c]$ (*e.g.*, $[x < y]$ has value 1 if $x < y$ and 0 otherwise). We employ the notation $\texttt{CMove}(z, x, a)$ to denote an oblivious subroutine that conditionally assigns the value of $a$ to $x$ if $z$ equals 1; if $z$ is 0, $x$ remains unmodified. The concrete implementations of the above operations could be based on assembly instructions [42] or branchless XOR-based C code [40]. We use $\perp$ to represent a dummy element or tuple, which is utilized solely for the purpose of padding and is designed to exert no influence on the outcome of the computation.

*Sorting* `OSort`. The bitonic sort [9] stands as the most favored oblivious sorting algorithm, celebrated for its simplicity and practicality. It accomplishes the sorting of $n$ elements in $O(n \log^2 n)$ time. While there exist oblivious sorting algorithms with lower asymptotic complexity [3, 7, 22], they are either non fully oblivious (assuming a super-constant sized trusted memory without obliviousness requirement), or encumbered by impractically large constant factors (outpace bitonic sort only when the input size is exceedingly large, which is an uncommon circumstance in the distributed setting). In this paper, we use $\texttt{OSort}(X, K)$ to represent an oblivious sorting operation that sorts $X$ by key $K$. Note that we will also discuss oblivious sorting under in the distributed setting that globally sorts the data across the servers. For disambiguation, we always use `OSort` to refer to the locally sorting in the standalone setting, while using "sorting" to refer to the globally sorting in the distributed setting.

*Compaction* `OCompact`. The compaction operator takes an array $X$ of $n$ elements and a binary array $M$ of length $n$ as input. The positions of $M$ with a value of 1 indicate "marked" elements, while positions with a value of 0 indicate "unmarked" elements. The compaction operation rearranges the elements in $X$ such that all marked items are positioned before unmarked items. We use $\texttt{OCompact}(X, M)$ to represent an oblivious compaction operation. Although it can be realized by invoking $\texttt{OSort}(X, M)$, we use the specialized oblivious algorithm for compaction in [47], which has only $O(n \log n)$ cost and is highly practical.

*Distribution* `ODistribute`. Let $X = (x_1, \ldots, x_n)$ be an array of non-dummy elements, and $T = (t_1, \ldots, t_n)$ is an array with distinct values such that $t_i \in [m]$ for all $i \in [n]$, where $m \geq n$ is a public parameter. The distribution operator `ODistribute` will output an array with size $m$ such that each $x_i$ is located at the $t_i$-th position for $i \in [m]$, while non-occupied positions are filled with dummy elements. Krastnikov et al. [32] proposed an oblivious distribution algorithm with $O(m \log m)$ cost under the constraint that elements in $T$ are in ascending order. Note that this constraint can be removed if we apply $\texttt{OSort}((X, T), T)$ in advance, and then the total cost of `ODistribute` will be $O(n \log^2 n + m \log m)$. We use $\texttt{ODistribute}(X, T, m)$ to represent an oblivious distribution operation.

*Partitioning* `OPartition`. Let $X = (x_1, \dots, x_n)$ be an array of elements, and $T = (t_1, \dots, t_n)$ is an array such that: (1) Each $t_i \in [p]$; (2) Let $X_j := \{i \in [n] \mid t_i = j\}$, then $|X_j| \le U$ for all $j \in [p]$. We define the partitioning operator `OPartition` as taking $X$ and $T$ as input, and outputs $p$ sequences $\{Y_j\}_{i=p}$, where each sequence $Y_j$ consists of $X_j$ and $U - |X_j|$ dummy elements, while the orders of them can be arbitrary. In other words, $Y_j$ is obtained by padding $X_j$ to the size bound $U$. In the distributed setting, `OPartition` is an important primitive for a server to reorganize its local data $X$ by their specified targets $T$, where each $t_i \in [p]$ is the designated server that $x_i$ should be sent to from this server. After `OPartition`, the $j$-th output sequence $Y_j$ will be sent to the $j$-th server. As the total size of the sequences is $pU$, the blow up factor of `OPartition` due to padding is $pU/n$. In this paper, to avoid oversized padding, our algorithms will always ensure that $U = O(n/p)$, so that the blow up factor is no more than a constant. SODA [37] has proposed an oblivious algorithm for `OPartition` (Algorithm 1 in [37]). The idea is to first `OSort` elements $X$ by $T$, compute the global position where each element should go to, and then apply `ODistribute` the elements to these positions. The complexity of their algorithm is hence $O(n \log^2 n)$.

## 3.2 Shuffle

We then return to the distributed setting. If any sequence $X$ in a distributed algorithms is physically distributed across the servers, we use $X[i]$ to denote the segment of $X$ located on the $i$-th server.

The most basic operator is *shuffle*. Note that in this paper, shuffle does not mean random permutation (a procedure that puts data items in a uniformly random order). Instead, it refers to the operator for re-distributing data across servers in the distributed setting, as adopted in distributed data analytics engines such as Apache Spark [58]. The shuffle operator takes two sequences $X$ and $T$ as input, where each $x \in X$ corresponds to a $t_x \in T$ which specifies the target server that $x$ should be sent to. Note that $X$ and $T$ locates across the $p$ servers, but each $(x, t_x)$ pair is in the same server. After the shuffle operator, the $j$-th server will receive $\{x \in X \mid t_x = j\}$, *i.e.*, all elements in $X$ with target $j$. For communication obliviousness, the shuffle operator also takes public parameters $\{U_i\}_{i=1}^{p}$ as input, and then the set of elements the $i$-th server receives will be padded to size $U_i$ by some dummy elements. Apparently, the shuffle operator can be implemented based on SODA's `OPartition`, as shown in Algorithm 1. Given $U_i = (1 + o(1))(n_i/p)$ for all $i$, the communication cost is $N$ and the computation cost of each server is $O(n \log^2 n)$.

We use "shuffle $X$ by $T$" to represent a shuffle operator with input $X$ and target servers $T$. Despite the standard definition, the shuffle operator also has two commonly used variants:

**Algorithm 1:** Shuffle

**Input:** $X$, $T$, and public parameters $\{U_i\}_{i=1}^{p}$

**for** $i \leftarrow 1$ **to** $p$ **do**
- $(Y_1, \dots, Y_p) \leftarrow \texttt{OPartition}(X[i], T[i], U_i)$;
- Send $Y_j$ to the $j$-th server for each $j \in [p]$;

*Random shuffle.* Random shuffle is a powerful operator that effectively eliminates the imbalance of the input. In the random shuffle operator, all $T$ are randomly chosen from $[p]$ uniformly and independently, *i.e.*, all elements will be randomly shuffled across the $p$ servers. Since each $t \in T$ is independent to the input, it could be safely publicized without breaching the obliviousness definition, hence no padding is required. Therefore, instead of calling `OPartition`, a random shuffle operator simply groups the elements with the same target server in a natural (non-oblivious) way (*e.g.*, by a length-$p$ array of lists). The computation cost can therefore be reduced to $O(n)$. We use "shuffle $X$ randomly" to represent a random shuffle operator with input $X$.

*Shuffle by key.* Assume each element is in the key-value form $x = (k, v)$ where $k$ is the key and $v$ is the value. The shuffle by key operator defines $t_x = h(k)$, where $h$ is a random oracle[1] that is public to all servers. The shuffle by key operator can gather elements with the same key across the servers to the same server for further computation. Since the targets of shuffle by key are data dependent, we should apply the `OPartition` for obliviousness. The parameters $\{U_i\}$ are determined by Theorem 3.1. We use "shuffle $X$ by key $K$" to represent a shuffle by key operator with input $X = (K, V)$ and its key $K$.

Theorem 3.1. *Setting $U_i = (1 + c_i)n_i/p$, if the keys of $X[i]$ are all distinct for any $i$, then the shuffle by key algorithm fails with probability at most $2^{-\sigma}$, where $c_i = \sqrt{2.08p(\sigma + 2\log p)/n_i} = o(1)$.*

In this paper, we omit all the proofs, which can be found in the full version of the paper [55].

## 3.3 Sorting

The sorting operator permutes the original input such that for each server $i$, its local data $X[i]$ is sorted, and for any two servers $i$ and $j$ where $i < j$, $x \le y$ for any $x \in X[i]$ and $y \in X[j]$. Please note that the sorting operator in this section is for the distributed setting, while `OSort` in Section 3.1 is for the standalone setting. Opaque [59] uses column sort [33], which is naturally oblivious.[2] Column sort requires four rounds of local sorting and communication, with the communication costs for these rounds being $N$, $N$, $N/2$, and $N/2$ respectively. Thus, the total communication cost sums up to $3N$. A recent study introduces DBucket [40], a distributed sorting algorithm. DBucket adapts the bucket oblivious random permutation proposed in [7] to the distributed setting. This is followed by a non-oblivious distribution sort [2] (*aka.* sample sort). The bucket random permutation leads to a communication cost of $2N$ due to

[1]A random oracle is an ideal function that maps distinct elements to independent and uniformly random outputs taken from its range. In practice we suggest using a cryptographic hash function such as BLAKE3.

[2]Opaque's sorting algorithm is not inherently computation oblivious; however, substituting its local sorts with `OSort` straightforwardly makes it computation oblivious.

padding, while the non-oblivious sort contributes an additional $N$. Consequently, the total communication cost for DBucket is also $3N$. We employ column sort in our experiments due to its simplicity.

## 3.4 Prefix Sum and Suffix Sum

Let $\oplus$ be a binary associative operator. The prefix sum operator takes $(x_1, \dots, x_N)$ as input and outputs $(x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \cdots \oplus x_N)$. The common choices of $\oplus$ are +, max, min, *etc.* If each $x_i$ is in the key-value pair form $x_i = (k_i, v_i)$, then $\oplus$ is usually defined as[3]

$$(k_1, v_1) \oplus (k_2, v_2) = \begin{cases} (k_2, v_1 \hat{\oplus} v_2) & \text{if } k_1 = k_2, \\ (k_2, v_2) & \text{otherwise,} \end{cases}$$

where $\hat{\oplus}$ is another binary associative operator that operates on the values. For example, in Opaque [59], the stage 2–3 of oblivious aggregate is essentially a prefix sum operator where the key is the set of grouping attributes and $\hat{\oplus}$ is the aggregate function, and the stage 2–3 of oblivious sort-merge join (PK join in our paper) is also equivalent to a prefix sum operator where the key is the set of join attributes and $\hat{\oplus}$ always returns the first input, *i.e.*, $v_1 \hat{\oplus} v_2 = v_1$.

The distributed algorithm for prefix sum operator [23] has communication cost $O(p)$, which is negligible compared with other operators as $p \ll N$. The algorithm is quite simple. First, each server $i$ locally computes the prefix sum on its input $X[i]$. Let $Y[i]$ be the output and $y_i$ be the last element of $Y[i]$, which is equal to sum of elements in $X[i]$. Each server sends $y_i$ to the first server, who then locally computes the prefix sum of $\{y_i\}_{i=1}^p$. Denote $\{z_i\}_{i=1}^p$ to be the output. The first server sends each $z_i$ to the $i+1$-th server for all $i \in [p-1]$. Finally each server $i \geq 2$ adds the element $z_{i-1}$ it receives to all the elements in $Y[i]$, *i.e.*, updates each $y \in Y[i]$ to $z_{i-1} \oplus y$. Then $(Y[i])_{i=1}^p$ is the prefix sum of $(X[i])_{i=1}^p$.

We will also need the *suffix sum* operator, which takes the same input as prefix sum but outputs $(x_1 \oplus x_2 \oplus \cdots \oplus x_N, x_2 \oplus \cdots \oplus x_N, \dots, x_{N-1} \oplus x_N, x_N)$. It is trivial to implement oblivious suffix sum algorithm as the prefix sum operator in a symmetric way with the same costs, and we omit the details.

## 3.5 Join

In database theory, a *join*[4] operator takes two tables $R$ and $S$ as input, and outputs the combinations of tuples from $R$ and $S$ that have the same values on the joined attributes (*aka.* join key). Without loss of generality, assume $R = R(A, B)$ and $S = S(B, C)$ and let the join key be $B$, then the join result of $R$ and $S$ is

$$R(A, B) \bowtie S(B, C) = \{(a, b, c) \mid (a, b) \in R \wedge (b, c) \in S\}.$$

In the rest of this paper, we denote $N_1 = |R|$, $N_2 = |S|$, and $M = |R \bowtie S|$. We define $\alpha_1$, the maximum degrees of the join key on $R$, as $\alpha_1 := \max_{b_0} |\{(a, b) \in R \mid b = b_0\}|$. Similarly, $\alpha_2 := \max_{b_0} |\{(b, c) \in S \mid b = b_0\}|$. We also define the $\ell_\infty$*-skewness* of a join with output size $M$ as $\phi = \alpha_1 \alpha_2 / M$.

*Comparison between oblivious joins.* The theoretical comparison between our oblivious join Jodes with existing ones is summarized in Table 2. The standalone algorithm works by all servers sending data to the first server who then performs state-of-the-art local oblivious join [32] in the standalone setting, splits the join result to $p$ parts, and sends each part to the corresponding server. *Cartesian join* first ignores the join conditions and computes the cartesian product of the two input tables [1, 11], and then filters the output tuples that does not meet the join conditions out by oblivious filter [37]. It is notable that Jodes has computation cost $O(1/p)$ of the standalone algorithm. The speed up factor $\Theta(p)$ means that Jodes has perfectly balanced the computation to the $p$ servers asymptotically.

**Table 2: Comparisons between oblivious join algorithms. Computation costs are presented asymptotically.**

| Algorithm | Communication | Computation |
|---|---|---|
| Standalone | $N + M$ | $p(n+m)\log^2 n$ |
| Cartesian join | $N_1 N_2$ | $p(n \log n)^2$ |
| SODA [37] | $4N + (p\phi + 1)M$ | $(n + (p\phi + 1)m)\log^2 n$ |
| Jodes (Ours) | $7N + 2M + \min(2M, Np)$ | $(n+m)\log^2 n$ |

SODA [37] proposed the first specialized oblivious algorithm for a general join. In addition to the total input size $N$ and the output size $M$, SODA's join algorithm also reveals $\alpha_1, \alpha_2$, the maximum degrees of the join key of the two tables. The key idea of SODA's join algorithm is to first arrange all the various-sized join groups into a set of equally-sized bins (first level assignment), and then distribute the bins to servers in a load balanced manner (second level assignment). Thereafter, each server computes the local join based on its assigned bins. To achieve obliviousness, the local join at each server produces an output of size $M/p + \alpha_1 \alpha_2$ padded with some dummy tuples. Optionally, the dummy tuples could be ultimately removed by SODA's filter algorithm. Note that in second level assignment, the granularity of the involved shuffle is bins, with numbers bounded by $O(N/(\alpha_1 + \alpha_2))$. As a result, it implicitly assumes $N = \Omega((\alpha_1 + \alpha_2)p^2\sigma)$ to avoid padding by a super-constant factor, which means that it is infeasible to set $\alpha_1, \alpha_2$ to the worst sizes $N_1, N_2$ to achieve the same security level as other algorithms, where $N_1$ and $N_2$ are the sizes of the two input tables respectively. In conclusion, Jodes has both costs asymptotically strictly better than all existing algorithms, except that when $\alpha_1 \alpha_2 = O(M/p)$, *i.e.*, $p\phi = O(1)$, SODA's join has the same complexity with Jodes. But in any case, SODA's join provides a weaker security guarantee than Jodes.

*3.5.1 Primary Key Join.* We consider a special type of join, *primary key join* (PK join), which guarantees that the join key is the primary key of $S$, *i.e.*, All tuples in $S$ have distinct values in $B$. With this constraint on $S$, PK join typically gains more efficient algorithm than general join. Opaque [59] supports oblivious PK join following the idea of sort-merge join. It calls the oblivious sorting operator twice on the union of the two tables, and the communication and computation costs of their algorithm are $6N_1 + 6N_2$ and $O(n \log^2 n)$ respectively, where $n = (N_1 + N_2)/p$. The oblivious join algorithm in SODA [37] can also be applied to PK join: By the primary key constraint, $\alpha_2 = 1$ and $M \leq N_1$, hence it has communication cost $5N_1 + 4N_2 + p\alpha_1$ and computation cost $O((n + \alpha_1)\log^2 n)$.

# 4 DESIGN

We propose the design of our algorithms in this section.

## 4.1 Shuffle

[3]To implement $\oplus$ in an oblivious way, one could run `CMove`$([k_1 = k_2], v_2, v_1 \hat{\oplus} v_2)$ and then simply outputs $(k_2, v_2)$.

[4]We only consider natural join (*aka.* equi-join) in this paper.

The computation cost of the shuffle algorithm (Algorithm 1) is primarily attributed to `OPartition`. We note that `OPartition` does not need elements in each output sequence to be sorted, hence employing `OSort` on all elements is superfluous. We borrow the idea of quicksort and propose our oblivious algorithm for `OPartition` as follows.[5] At the high level, quicksort is a recursive algorithm that partitions data to several buckets, where any element in $i$-th bucket is not larger than any element in the $j$-th bucket for any $i < j$. Then it applies the quicksort algorithm on each bucket recursively. In `OPartition`, this recursion can be early stopped as long as the bucket size is at most $U$, hence the number of recursion levels can be reduced from $\log n$ to $\log p$. For each level, we choose the middle point as the pivot, and partition the elements to two parts according to the pivot by using `OCompact`: Move all elements $(x_i, t_i)$ with $z_i = 1$ in front of other elements in an oblivious way. Such $z_i$ could be determined by a linear scan. Afterwards, we input each of the two parts recursively. The cost of `OCompact` in each level is $O(n \log n)$ and the number of levels is $\lceil \log p \rceil$, so the total cost is $O(n \log n \log p)$. Hence our algorithm reduces the computation cost of the shuffle operator from SODA's $O(n \log^2 n)$ to $O(n \log n \log p)$.

## 4.2 Primary Key Join

Below we present our oblivious PK join algorithm (Algorithm 2) with lower costs. Our basic idea follows the aggregate algorithm in SODA [37] that tuples with the same key should be shuffled to the same server so that they can be joined locally. The main issue of simply invoking the shuffle by key operator is that it requires the tuples of the input table are distinct on their keys (Theorem 3.1), which holds for $S$ but not for $R$. To resolve this issue, for tuples in $R$ with the same key in each server, we choose one of them as the *representative* and mark other tuples as *inactive*. In the shuffle by key operator, the representatives are shuffled by the join key while the inactive tuples are shuffled to random target servers independently. Then we can join the representatives of $R$ with all tuples of $S$ locally in each server (Line 12–22). Taking the information from $S$, the representatives then go back to their original servers (Line 23) and distribute the data they receive from $S$ to those inactive tuples (Line 25–28). Note that in Line 8, representatives have distinct $B$ but with $Z = 0$ while inactive tuples have distinct and nonzero $Z$, so they are all distinct on $(B, Z)$, hence shuffle by key operator can be applied. Also note that Line 23 is essentially the reverse of the shuffle in Line 8, so they should have the same padding size. Our algorithm has communication and computation cost $2N_1 + N_2$ and $O(n \log^2 n)$ respectively.

*Example 4.1.* Consider the example as shown in Figure 1, in which there are two servers S1 and S2. The representatives of $R$ in S1 and S2 are $(a_1, 1)$, $(a_2, 2)$ and $(a_2, 1)$, $(a_1, 2)$ respectively. All other tuples are deemed inactive and represented in gray. Step (a) is the shuffle by key operation, during which representatives in $R$ and all tuples in $S$ are shuffled by their $B$ values ($h(2) = h(3) = 1$ and $h(1) = h(4) = 2$), whereas the inactive tuples are randomly assigned to a server. Step (b) entails executing a local PK join on each server. Note that inactive tuples are excluded from this join and instead have their $C$ values designated as $\perp$ (dummy). In step (c), all tuples are shuffled back to their original server. For instance,

[5]The algorithmic description is in the full version of this paper [55].

**Algorithm 2:** Oblivious PK join

**Input:** $R(A, B)$ and $S(B, C)$ where $B$ is the primary key of $S$
**Output:** $V(A, B, C) = R \bowtie S$

1 Add column $Z$ to $R[i]$ with $Z \leftarrow 0$;
2 **for** $i \leftarrow 1$ **to** $p$ **do**
3     `OSort`$(R[i], B)$;
4     Add column $I$ to $R[i]$ with $I \leftarrow i$ ; // Record the original server id
5     **for** $j \leftarrow 2$ **to** $|R[i]|$ **do**
6         $(t_{j-1}, t_j) \leftarrow$ the $(j-1, j)$-th tuple of $R[i]$;
7         `CMove`$([t_j.B = t_{j-1}.B], t_j.Z, j)$ ; // Set inactive tuples to distinct and positive $Z$
8 Shuffle $R$ by key $(B, Z)$ ; // Inactive tuples are randomly shuffled
9 Shuffle $S$ by key $(B, 0)$;
10 Initialize table $V(A, B, C, I, Z)$;
11 **for** $i \leftarrow 1$ **to** $p$ **do**
12     $n_0 \leftarrow |R[i]|$;
13     Add column $C$ to $R[i]$;
14     Add columns $A, I, Z$ to $S[i]$ with $Z \leftarrow -1$;
15     $V[i] \leftarrow R[i] \cup S[i]$;
16     `OSort`$(V[i], (B, Z))$;
17     **for** $j \leftarrow 2$ **to** $|V[i]|$ **do**
18         $(t_{j-1}, t_j) \leftarrow$ the $(j-1, j)$-th tuple of $V[i]$;
19         $c \leftarrow [t_j.B = t_{j-1}.B \wedge t_j.Z = 0]$;
20         `CMove`$(c, t_j.C, t_{j-1}.C)$;
21     `OCompact`$(V[i], [Z \geq 0])$ ; // Move tuples from $R$ to the front
22     Truncate $V[i]$ to size $n_0$;
23 Shuffle $V = (V[1], \ldots, V[p])$ by $V.I$ ; // Shuffle tuples back
24 **for** $i \leftarrow 1$ **to** $p$ **do**
25     `OSort`$(V[i], (B, Z))$;
26     **for** $j \leftarrow 2$ **to** $|V[i]|$ **do**
27         $(t_{j-1}, t_j) \leftarrow$ the $(j-1, j)$-th tuple of $V[i]$;
28         `CMove`$([t_j.B = t_{j-1}.B], t_j.C, t_{j-1}.C)$;
29 Remove columns $I, Z$ from $V$;
30 **return** $V$;

| | $R(A,B)$ | $S(B,C)$ | | $R(A,B)$ | $S(B,C)$ | | $T(A,B,C)$ | | $T(A,B,C)$ | | $T(A,B,C)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| S1 | $(a_1,1)$, $(a_2,2)$, $(a_3,2)$, $(a_5,2)$ | $(1,c_1)$, $(2,c_2)$ | $\xrightarrow{(a)}$ | $(a_1,2)$, $(a_2,2)$, $(a_3,1)$, $(a_5,2)$ | $(2,c_2)$, $(3,c_1)$ | $\xrightarrow{(b)}$ | $(a_1,2,c_2)$, $(a_2,2,c_2)$, $(a_3,1,\perp)$, $(a_5,2,\perp)$ | $\xrightarrow{(c)}$ | $(a_1,1,c_1)$, $(a_2,2,c_2)$, $(a_3,2,\perp)$, $(a_5,2,\perp)$ | $\xrightarrow{(d)}$ | $(a_1,1,c_1)$, $(a_2,2,c_2)$, $(a_3,2,c_2)$, $(a_5,2,c_2)$ |
| S2 | $(a_2,1)$, $(a_3,1)$, $(a_1,2)$, $(a_4,2)$ | $(3,c_1)$, $(4,c_3)$ | | $(a_1,1)$, $(a_2,1)$, $(a_3,2)$, $(a_4,2)$ | $(1,c_1)$, $(4,c_3)$ | | $(a_1,1,c_1)$, $(a_2,1,c_1)$, $(a_3,2,\perp)$, $(a_4,2,\perp)$ | | $(a_2,1,c_1)$, $(a_3,1,\perp)$, $(a_1,2,c_2)$, $(a_4,2,\perp)$ | | $(a_2,1,c_1)$, $(a_3,1,c_1)$, $(a_1,2,c_2)$, $(a_4,2,c_2)$ |

**Figure 1: PK join algorithm example**

the tuple $(a_3, 2, \perp)$, which was initially $(a_3, 2)$ on S1, is relocated back to S1. Finally, in step (d), the active tuples distribute their $C$ values to the inactive tuples, thus completing the PK join process.

## 4.3 Expansion

Before presenting our join algorithm, we need to introduce the expansion operator first. Given a public parameter $M$, the expansion operator takes two arrays $X = (x_1, \ldots, x_N)$ and $D = (d_1, \ldots, d_N)$ as input, where each $d_i$ is a non-negative integer and $d_\perp := M - \sum_{i=1}^{N} d_i \geq 0$. The values $d_i$ indicates the number of repetitions that

$x_i$ should appear in the output, *i.e.*, the output is a length-$M$ array:

$$(\underbrace{x_1, \ldots, x_1}_{d_1 \text{ times}}, \underbrace{x_2, \ldots, x_2}_{d_2 \text{ times}}, \ldots, \underbrace{x_N, \ldots, x_N}_{d_N \text{ times}}, \ldots, \underbrace{\perp, \ldots, \perp}_{d_\perp \text{ times}}).$$

Note that those $x_i$ with $d_i = 0$ would not appear in the output. The expansion operator was initially proposed for database join in [6] and the oblivious standalone algorithm is formally described in [32]. In this section, we propose the first oblivious algorithm for the expansion operator in the distributed setting. Each server holds $N/p$ elements of $X$ and $D$ as input, and will hold $M/p$ elements of $Y$ as output after computation. Our algorithm is described in Algorithm 3, in which we (logically) organize the input as a table $R(X, D)$ and output as a table $S(X)$ for better readability.

Our algorithm works in two steps. Assume the output array $\{y_i\}_{i=1}^M$ is initially a length-$M$ array filled with dummy elements, *i.e.*, $y_i = \perp$ for all $i \in [M]$. Note that the largest index of each $x_i$ appearing in the output array is supposed to be $l_i := \sum_{j=1}^{i} d_j$, except that those with $d_i = 0$ would not appear. The first step is to set $y_{l_i}$ to $x_i$ for each $i \in [N]$ for those $d_i > 0$, and the second step is to replace each dummy element with the first non-dummy element after it (if there is), which could be realized by a suffix sum operator by defining proper $\oplus$ (Line 14 in Algorithm 3).

To achieve the first step obliviously, we first note that the array $\{l_i\}_{i=1}^N$ can be obtained by calling a prefix sum operator with input $(d_1, \ldots, d_N)$. Since each server will hold $m$ elements of the output array, the $l_i$-th element in the output will be held by server $t_i = \lceil l_i/m \rceil$, which suggests we should shuffle each $x_i$ to the $t_i$-th server (if $d_i = 0$, $x_i$ is simply ignored). We denote this shuffle `SF1`. Since the target servers in `SF1` are data dependent, it needs padding to achieve obliviousness. We perform a random shuffle `SF0` before `SF1` to balance the data, so that the padding size of `SF1` is bounded.

Theorem 4.2. *Let* `SF0` *be the random shuffle and* `SF1` *be the shuffle following* `SF0`*. If we set $U_i = (1 + c_i)n_i \cdot \min(m/N, 1)$ in* `SF1`*, then Algorithm 3 has communication cost $N + \min(M, Np)$ and computation cost $O(m \log n + \min(m, N) \log^2 n)$ with failure probability at most $2^{-\sigma}$, where $n_i$ is the number of tuples on the $i$-th server after* `SF0`*, and $c_i = \sqrt{2.08 \max(N/m, 1)(\sigma + 2 \log p)/n_i} = o(1)$.*

*Example 4.3.* Consider the example in Figure 2 with $p = 3$, $M = 18$, and $d_\perp = 2$. Each server will hold $m = M/p = 6$ elements of the output. Our algorithm first computes the prefix sum of $(1, 3, 1, 0, 5, 2, 1, 1, 2)$ as $L$ in step (a), indicating that $x_i$ will lastly appear at the $l_i$-th location in the output array, except that $d$ will not appear. This in turn implies $x_i$ will lastly appear in the $p_i$-th location of the $t_i$-th server, with $T$ and $P$ locally computed in step (b). In step (c), we shuffle $R$ randomly, then shuffle it with target servers of tuples specified by $T$ with proper padding. Afterwards, each server locally put each tuple $t$ at the $t.P$-th position in its server by `ODistribute`. The result is shown as the fourth table in this figure. Step (d) is a suffix sum operation as described in Line 14 of Algorithm 3 which finally yields the expansion result.

## 4.4 Oblivious Join

Now we are ready to present our oblivious join algorithm. Note that our idea for PK join is not directly applicable to a generalized join operation, because for any $b$, there could be multiples tuples

---

**Algorithm 3:** Oblivious expansion

**Input:** $R(X, D)$, and public parameter $M$

**Output:** $S(X)$ where $t.X$ appears $t.D$ times for any $t \in R$

```
1  Add columns (L, T, P) to R;
2  R.L ← the prefix sum of R.D ;                    // Target global position
3  for i ← 1 to p do
4      for j ← 1 to |R[i]| do
5          t_j ← the j-th tuple of R[i];
6          t_j.T ← ⌈t_j.L/m⌉ ;                            // Target server
7          t_j.P ← t_j.L − (t_j.T − 1)m ; // Target position in the target
             server
8          CMove([t_j.D = 0], t_j, ⊥);
9  Shuffle R randomly ;                                    // No padding
10 Shuffle R by R.T with padding sizes specified by Theorem 4.2;
11 Initialize table S(X);
12 for i ← 1 to p do
13     S[i].X ← ODistribute(R[i].X, R[i].P, m);
14 Run suffix sum operator on S with ⊕ defined as: x_1 ⊕ x_2 is x_2 if
     x_1 = ⊥, otherwise x_1;
15 return S;
```

| | $R(X,D)$ | | $R(X,D,L)$ | | $R(X,T,P)$ | | $S(X)$ | | $S(X)$ |
|---|---|---|---|---|---|---|---|---|---|
| S1 | $(a,1)$, $(b,3)$, $(c,1)$ | | $(a,1,1)$, $(b,3,4)$, $(c,1,5)$ | | $(a,1,1)$, $(b,1,4)$, $(c,1,5)$ | | $a, \perp, \perp$, $b, c, \perp$ | | $a, b, b$, $b, c, e$ |
| S2 | $(d,0)$, $(e,5)$, $(f,2)$ | $\xrightarrow{(a)}$ | $(d,0,5)$, $(e,5,10)$, $(f,2,12)$ | $\xrightarrow{(b)}$ | $\perp$, $(e,2,4)$, $(f,2,6)$ | $\xrightarrow{(c)}$ | $\perp, \perp, \perp$, $e, \perp, f$ | $\xrightarrow{(d)}$ | $e, e, e$, $e, f, f$ |
| S3 | $(g,1)$, $(h,1)$, $(i,2)$ | | $(g,1,13)$, $(h,1,14)$, $(i,2,16)$ | | $(g,3,1)$, $(h,3,2)$, $(i,3,4)$ | | $g, h, \perp$, $i, \perp, \perp$ | | $g, h, i$, $i, \perp, \perp$ |

**Figure 2: Expansion algorithm example with $M = 18$**

in both $R$ and $S$ with $B = b$. While it is still feasible to select any tuple from $R$ with $B = b$ as a representative, it is not known how to efficiently associate all corresponding tuples in $S$ with $B = b$ to this chosen representative.

We start by revisiting the state-of-the-art standalone oblivious join algorithm [32]. The high level idea is based on the observation that for any $(a, b) \in R$, it appears $\deg_S(b)$ times in the join result, where $\deg_S(b)$ is the number of tuples in $S$ with $B = b$, which we call the *degree* of $b$ in $S$. These degrees can be computed by combining sorting and prefix sum operators, and can be attached to the correct tuples in $R$ by a PK join operator. Then an expansion operator expands $R$ according to the degree of $S$, increasing the total size to $M$, the output table size. Note that this expansion requires $M$ as introduced in Section 2.5. These steps are then applied to $S$ symmetrically. Finally, it aligns the two expanded tables properly by an extra sorting on $S$ by join key and its *alignment key*, which could be computed by the degrees of the two tables. Note that the above algorithm is essentially the composition of the sorting, aggregate, PK join, and expansion operators. By instantiating these operators with our proposed distributed oblivious algorithms, it is transformed to the distributed version correctly.

Our oblivious join algorithm Jodes is presented in Algorithm 4, and the subroutine that computes the degrees of the two tables are described in Algorithm 5. Despite following the idea of the standalone oblivious join algorithm, we also optimize Jodes in distributed setting by noting that the final alignment can be implemented without the sorting operator. Specifically, the original alignment key $L$ indicates the positions of the tuples in each group of $B$. We redefine $L$ so that it indicates the global positions, which are computed as in Line 7–13. Instead of simply performing global sorting on $L$, we first compute the target servers $T$ of the tuples by the alignment key $L$ (Line 14), shuffle the table by $T$, and then perform `OSort` on the alignment key in each server. However, the target servers are data dependent, hence padding is required. Similar to our expansion algorithm, we perform a random shuffle in advance to balance the data, and setting padding size as stated in Theorem 4.4 is adequate. The communication costs of the first 6 lines are $3N_1+3N_2, 2N_1+N_2, 0, N_1+\min(M, N_1p), N_1+2N_2, N_2+\min(M, N_2p)$ respectively, and the communication cost of the each of the two shuffles is $M$. Other operators involve only costs with low-order term. Hence the total communication cost of Jodes is $7(N_1+N_2)+\min(M, N_1p)+\min(M, N_2p)+2M \leq 7N+2M+\min(2M, Np)$ where $N = N_1+N_2$. The computation cost of Jodes is dominated by `OSort` before and after expansion, which is $O((n+m)\log^2 n)$.

Algorithm 4 assumes $M$ is a public parameter. If the padding scheme is "no padding", *i.e.*, $M$ is the exact output size of the join as in SODA [37], then we can simply compute $M$ by summing all the degrees that $R$ receives after PK join, without the need of it being public. Specifically, insert "$M \leftarrow$ sum of $R'.D_S$" after Line 2.

Theorem 4.4. *Let* `SF0` *be the random shuffle (Line 15) and* `SF1` *be the other shuffle (Line 16) in Algorithm 3. If we set* $U_i = (1+c_i)n_i/p$ *in* `SF1`*, then it fails with probability at most* $2^{-\sigma}$*, where* $n_i$ *is the number of tuples on the* $i$*-th server after* `SF0`*, and* $c_i = \sqrt{2.08p(\sigma+2\log p)/n_i} = o(1)$.

*Example 4.5.* Consider the example shown in Figure 3, in which the output size bound $M$ is set to the true join size, *i.e.*, no padding. The subroutine Algorithm 5 corresponds to step (a), which includes two sub-steps: (a1) computing the prefix sum on key-value pair $(B, 1)$ to get $D_R$, and (a2) updating $D_R$ by suffix max and then obtaining $D_S$ by PK join. Step (b) is to apply the expansion operator on the degree of the other table. Besides, for $\bar{S}$, it also computes the alignment key $L$ and the target servers $T$. In step (c), we apply the two shuffle operators and then local `OSort` so that $\bar{S}$ is ordered by $L$. The final step (d) is to combine $\bar{R}$ and $\bar{S}$ to get the join result $V$.

## 5 SECURITY ANALYSIS

In this section, we prove that our proposed algorithms are both communication oblivious and computation oblivious. Recall Definition 2.1 for communication obliviousness. Note that all our algorithms involve communication only by calling the shuffle operator accompanied by determinate padding sizes, and the servers will receive messages whose sizes are congruent with these padding sizes $\{U_i\}$, which can be computed from the input sizes $\{n_i\}$ and public parameters $p$ and $\sigma$ (Theorem 3.1, 4.2, 4.4). Therefore, let the simulator simply outputs $\bar{S}$ as random numbers with sizes $\{U_i\}$, then the transcript of the algorithm and $\bar{S}$ are in identical sizes and hence indistinguishable.

For computation obliviousness in Definition 2.2, note that none of our algorithms involves any data dependent operations due to: (1) the execution of all loops with publicly known sizes; (2) the substitution of all conditional branches with `CMove` instructions; (3) the utilization of data independent memory access locations; (4) the employment of primitives that are intrinsically oblivious, *e.g.*, `OSort` and `OCompact`. Therefore, the simulator can simply simulate the memory access pattern by running the algorithm with arbitrary input (of the same sizes), and the adversary could not distinguish between the access patterns from the true input and the simulated input. Note that for the random shuffle operator, the access pattern is random, but the distribution of the access pattern is data

---

**Algorithm 4:** Oblivious join Jodes

**Input:** $R(A, B)$ and $S(B, C)$; public output size bound $M$
**Output:** $V(A, B, C) = R(A, B) \bowtie S(B, C)$

1 Sort both $R$ and $S$ by $B$;
2 $R'(A, B, D_R, D_S) \leftarrow$ run Algorithm 5 with input $R, S$;
3 Remove column $D_R$ from $R'$;
4 $S'(B, C, D_R, D_S) \leftarrow$ run Algorithm 5 with input $S, R$;
5 $\bar{R}(A, B) \leftarrow$ expansion with input $(R'.A, R'.B)$, $(R'.D_S)$ and $M$;
6 $\bar{S}(B, C, D_R, D_S) \leftarrow$ expansion with input $S'$, $S'.D_R$ and $M$;
7 Add column $I, J, L, T$ to $\bar{S}$;
8 $\bar{S}.I \leftarrow$ prefix sum on key-value pair $(B, 1)$;
9 $\bar{S}.J \leftarrow$ prefix min on key-value pair $(B, [M])$;
10 **for** $i \leftarrow 1$ **to** $p$ **do**
11     **for** $t \in \bar{S}[i]$ **do**
12         $q \leftarrow t.I - 1$;
13         $t.L \leftarrow \lfloor q/t.D_R \rfloor + (q \bmod t.D_R) \cdot t.D_S + t.J$;
14         $t.T \leftarrow \lceil t.L/m \rceil$;
15 Shuffle $\bar{S}$ randomly;
16 Shuffle $\bar{S}$ by $S.T$ with padding size specified by Theorem 4.4;
17 Initialize table $V(A, B, C)$;
18 **for** $i \leftarrow 1$ **to** $p$ **do**
19     `OSort`$(\bar{S}[i], L)$;
20     **for** $j \leftarrow 1$ **to** $m$ **do**
21         $(t_R, t_S) \leftarrow$ the $j$-th tuple of $(\bar{R}[i], \bar{S}[i])$;
22         Insert $(t_R.A, t_R.B, t_S.C)$ to $V[i]$;
23 **return** $V$;

---

**Algorithm 5:** Compute degrees

**Input:** $R(A, B)$ and $S(B, C)$, both ordered by $B$ ;
**Output:** $R(A, B, D_R, D_S)$

Add column $D_R$ to $R$;
$R.D_R \leftarrow$ prefix sum on key-value pair $(B, 1)$;
$R.D_R \leftarrow$ suffix max on key-value pair $(B, D_R)$, *i.e.*, $\oplus$ is defined as $x \oplus y = \max(x, y)$;
Add column $D_S$ to $S$;
$S.D_S \leftarrow$ prefix sum on key-value pair $(B, 1)$;
$S.D_S \leftarrow$ suffix max on key-value pair $(B, D_S)$;
**for** $i \leftarrow 1$ **to** $p$ **do**
    **for** $j \leftarrow 2$ **to** $|S[i]|$ **do**
        $(t_{j-1}, t_j) \leftarrow$ the $(j-1, j)$-th tuple of $S[i]$;
        `CMove`$([t_{j-1}.B = t_j.B], t_{j-1}, \perp)$; // Remove duplicates
$R(A, B, D_R, D_S) \leftarrow R(A, B, D_R) \bowtie S(B, C, D_S)$ ; // PK join with communication cost $2|R| + |S|$
**return** $R$;

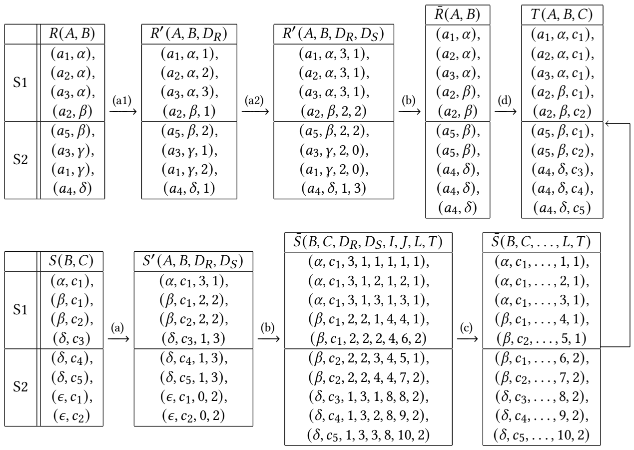



**Figure 3: Join algorithm example**

independent, thereby excluding any possibility for the adversary to differentiate.

# 6 EVALUATION

## 6.1 Experimental Setup

*Environment.* We deployed the distributed environment on 16 machines, each equipped with an Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz and having 1TB RAM capacity. For each machine, we initialized the enclave with 64GB EPC size. The machines were interconnected via a local area network with bandwidth up to 2.9 GB/s, facilitating communication using the HTTP protocol built on Facebook's Proxygen framework [20]. We counted the communication cost as the total number of bytes sent across the servers. We chose AES-GCM with 128-bit key as the data encryption scheme, with encryption and decryption speed (inside enclave) around 1.0GB/s. The compiler was GCC 8.5.0 with "-O3" optimization enabled, and the implementation of `CMove` followed the XOR-based C code in [40]. We enabled multi-threading outside the enclave, *i.e.*, a server may receive data from other servers and perform computations inside the enclave simultaneously, but all computations within the enclave were executed using a single thread.

*Default settings.* For our experiments, we standardized the computational environment by configuring the number of servers to $p = 16$ and setting the security parameter to $\sigma = 40$. For join operator, we set $M$ to be the output size (no output padding). For a fair comparison, in evaluations involving shuffle or PK join operations, except when directly comparing these primitives, we consistently employed our implementations as described in Section 3. This ensured that any observed performance differences could be attributed to the intrinsic merits of the algorithms under investigation rather than variations in the underlying primitives.

## 6.2 Performance of Basic Operators

*OPartition.* Our first improvement to the baseline is the standalone algorithm for `OPartition`, which is the key building block for the shuffle operator. We benchmarked the local computation phase of our algorithm against that of SODA [37] using inputs generated randomly, varying input size $N$, the number of servers $p$, or the security parameter $\sigma$. The nature of obliviousness ensures that the performance of the algorithms remains consistent regardless of the variability in the input data. The results are shown in Figure 4. Our algorithm shows a substantial empirical performance improvement, ranging from 60% to as much as 310%. Notably, this enhancement factor grows in proportion to the increase in input size $N$ or the decrease in the number of servers $p$. This trend is in line with our theoretical analysis, which states an improvement factor on the order of $\log n/\log p$. Besides, we find that both algorithms are insensitive to the security parameter. In applications requiring even smaller failure probability, the performance degradation will be minimal.

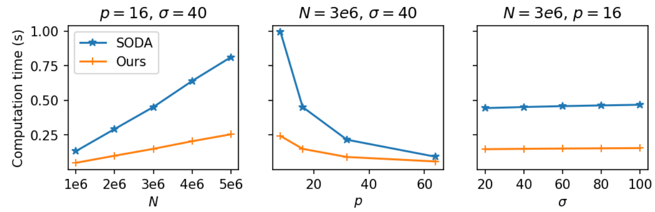



**Figure 4: Computation time of `OPartition` varying input size $N$, number of servers $p$, or security parameter $\sigma$.**

**Table 3: PK join input table information; Total time and communication cost of the PK join algorithms**

| | `customer` | | `orders` | | | |
|---|---|---|---|---|---|---|
| #Rows | $N_2 = 1.5\times 10^6$ | | $N_1 = 1.5\times 10^7$ | | | |
| Zipf parameter $z$ | 0 | | 0 | 0.5 | 1 | 1.5 |
| Max degree $\alpha$ | 1 | | 37 | 449 | 50697 | 210490 |
| | Ours | Opaque | SODA | | | |
| #Output ($\times 10^7$) | 1.50 | 1.50 | 1.50 | 1.50 | 1.58 | 1.84 |
| Total time (s) | 6.44 | 12.5 | 11.0 | 11.0 | 11.3 | 12.0 |
| Comm. cost (GB) | 0.62 | 1.56 | 1.19 | 1.19 | 1.23 | 1.34 |

*Primary key join (PK join).* We conducted the PK join experiments on the well-known TPC-H dataset evaluating our Algorithm 2 against Opaque's PK join and SODA's join using the query below:

```
SELECT * FROM orders JOIN customer ON o_custkey = c_custkey;
```

Note that `c_custkey` is the primary key of table `customer`, so the query is a PK join. Both tables were generated by the code from a publicly accessible GitHub repository [57] with a scale factor of 10 and with the `orders` table exhibiting four distinct values of Zipfian distribution parameters, denoted by $z$, as outlined in Table 3. Irrelevant columns were eliminated from computation, leaving only `c_custkey`, `c_nationkey`, `o_orderkey`, and `o_custkey`.

The results are also in Table 3. Different values of $z$ result in different maximum degrees on the join key, thereby affecting the output size of SODA's join but not impacting our algorithm or Opaque's join. In terms of overall running time, our algorithm improves upon the baselines by at least 70%. This improvement rises to more than 90% regarding communication costs. While SODA's join slightly outperforms Opaque's PK join, the gap narrows as $z$ increases, due to the increasing output size. Meanwhile, both Opaque's PK join and our algorithm maintain steady performance despite varying $z$ due to obliviousness.

**Table 4: Join input table information; Output sizes of the join algorithms**

| | DBLP | email | Youtube | wiki |
|---|---|---|---|---|
| #Input $N_1 = N_2$ | $1.0\times10^6$ | $4.2\times10^5$ | $2.9\times10^6$ | $2.9\times10^7$ |
| Max #dst $\alpha_1$ | 306 | 930 | 28576 | 3907 |
| Max #src $\alpha_2$ | 113 | 7631 | 4256 | 238040 |
| $\ell_\infty$-skewness $\phi$ | 0.0049 | 0.14 | 0.64 | 0.36 |
| #Output $M$ | $7.1\times10^6$ | $5.0\times10^7$ | $1.9\times10^8$ | $2.6\times10^9$ |
| #Output (SODA) | $7.7\times10^6$ | $16\times10^7$ | $21\times10^8$ | $17\times10^9$ |

## 6.3 Performance of Join

For join, in addition to evaluating Jodes and SODA, we also conducted tests on a single server using the state-of-the-art oblivious standalone join [32], in which all data is sent to the first server that performs the standalone join locally and then sends the results to the other servers.

*Varying datasets.* We evaluated the join operator on Stanford Large Network Dataset Collection [34]. We selected four graphs with various $\ell_\infty$-skewnesses: "com-DBLP" (**DBLP**), "email-EuAll" (**email**), "com-Youtube" (**Youtube**), and "wiki-topcats" (**wiki**). More information of the four graphs are in Table 4. Each graph was converted into a relational table format with two columns, `src` and `dst`, to represent the source and destination nodes of each edge. We assessed the performance on the following self-join query designed to identify all length-2 paths within the graphs:

```
SELECT * FROM graph R JOIN graph S ON R.dst = S.src;
```

Figure 5a includes the performance results, with the y-axis represented on a logarithmic scale. For the **wiki** dataset, both SODA and standalone algorithm could not complete within an hour. The speed-up of Jodes compared to the standalone algorithm ranges from 4x (for small data) to 6x (for large data). In contrast, our speed-up over SODA is highly dependent on the value of $\phi$: it is 1.1x for **DBLP** (small $\phi$), 1.6x for **email** (medium $\phi$), and 6x for **Youtube** (large $\phi$). Specifically, for the **Youtube** dataset, the total time of SODA using 16 servers even exceeds that of the standalone algorithm with only one server, thus losing the advantages of distribution. With respect to communication costs, the standalone algorithm incurs the least, equivalent to only the I/O size. Jodes exhibits slightly higher communication costs than SODA for **DBLP** and **email**, but lower costs for **Youtube**.

*Varying bandwidths.* We note that for **DBLP** and **email** in Figure 5a, Jodes has a shorter running time but incurs a higher communication costs than SODA. The reason is that SODA adopts complex design of local computations to minimize the communication cost. To see whether SODA outperforms Jodes for limited bandwidth, we reran the tests on the **email** dataset and limited the bandwidth to assess its impact on performance. The results are shown in Figure 5b. Since varying bandwidths does not change the communication costs, we only plot the running time. It appears that only under very limited bandwidth conditions (less than 25 MB/s), Jodes's performance is surpassed by SODA. However, we argue that in distributed settings, a large bandwidth requirement is reasonable. For example, all instances of Amazon EMR [48] have a minimum bandwidth of 10 Gbps (1280 MB/s).

*Varying I/O sizes.* We also conducted experiments on sampled data from the **wiki** dataset to assess the scalability of Jodes and SODA. Specifically, we sampled each row of the input table with probability $\epsilon$ independently and then performed the join on the sampled table, as described by the following SQL:

```
WITH sampled AS (SELECT * FROM graph WHERE rand() < ϵ)
SELECT * FROM sampled R JOIN sampled S ON R.dst = S.src;
```

Note that a sampling probability of $\epsilon$ induces the expected join size of the sampled table to be $\epsilon^2$ times that of the original table. We tried $\epsilon \in \{0.2, 0.4, 0.6, 0.8\}$, and the performance results are shown in Figure 5c. For $\epsilon = 0.8$, the standalone algorithm could not finish in an hour. The total time of all algorithms scales almost linearly to the I/O sizes, *i.e.*, the total sizes of input and output. The speed-up factor of Jodes to SODA ranges from 2.5 ($\epsilon = 0.2$) to 3.9 ($\epsilon = 0.8$). Jodes also incurs less communication cost, except when $\epsilon = 0.2$, where it is slightly higher.

*Varying number of servers.* To examine how the number of servers influences performance, we conducted experiments with different values of $p$ using the **Youtube** dataset, and the results are presented in Figure 5d. We observe that SODA does not scale well for large $p$: the running time when using 12 servers is almost identical to that when using 16 servers. The major reason is that the output size of SODA on each server is $M(1/p + \phi)$. Since the extra padding size $M\phi$ is independent of $p$ and dominates the join size $M/p$, increasing the number of servers does not significantly reduce the workload on each server, while the increasing total communication can even degrade the performance. On the contrary, the output size of Jodes on each server is always $M/p$, allowing it to scale much more effectively.

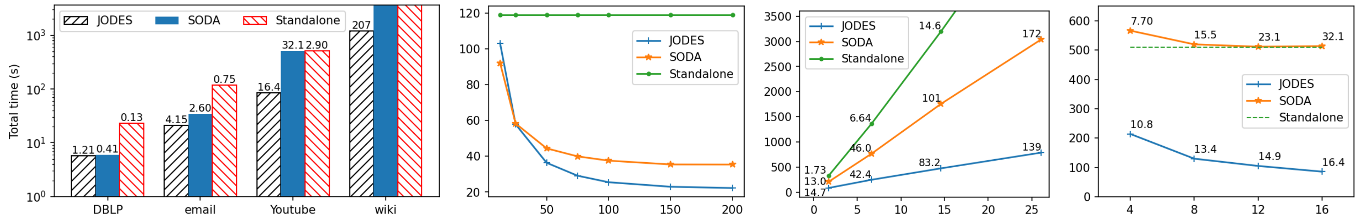
*Running time breakdown analysis.* Jodes (Algorithm 4) works in a manner similar to the standalone algorithm at a high level, which can be segmented into the following three phases:

(1) *Preparation.* This phase involves computing the degrees of join keys in both input tables and matching them through a primary key (PK) join (Lines 1–4);
(2) *Expansion.* In this phase, both tables are expanded from size $N$ to $M$ through expansion operations (Lines 5–6);
(3) *Alignment.* This phase focuses on aligning the two tables to ensure that the correct tuples match (Lines 7–23).

We evaluated the breakdown of running time for these three phases using the **YouTube** dataset, and the results are presented in Table 5. The findings align with our theoretical analysis, indicating that the preparation phase depends solely on $N$, while both the expansion and alignment phases are influenced by $M$, where $M \gg N$. Utilizing 16 servers, the performance improvement factors of Jodes compared to the standalone algorithm for the three phases are 5.2, 3.5, and 7.7, respectively. This suggests that optimizing the design of the expansion algorithm could be a productive pathway for further enhancing Jodes.

## 7 RELATED WORK

Existing analytic systems based on TEE include [4, 18, 24, 36, 50, 52, 59]. However, most of them only focus on the standalone setting. Ohrimenko et al. [41] firstly pointed out the leakage by the network traffic in the distributed setting. Their empirical analysis on datasets




**(a) Varying input datasets** **(b) Varying bandwidths (MB/s)** **(c) Varying I/O sizes (GB)** **(d) Varying number of servers**

**Figure 5: Total time of join, where the labels on top of the bars or adjacent to the data points are communication costs (GB).**

**Table 5: Running time breakdown of JODES and the standalone algorithm**

| | Preparation | Expansion | Alignment | Total |
|---|---|---|---|---|
| JODES | 5.7% | 37.2% | 57.1% | 100% |
| Standalone | 29.5% | 129.7% | 437.9% | 597% |

that include personal and geographical data shows that the runs of typical jobs can infer precise information about their input. To prevent such leakage, they provided a shuffle-in-the middle solution: before sending data (with some padding techniques) to the intended destination, permuting the input randomly among the servers in advance to remove potential skewness; Chan et al. [13] proposed a different solution based on oblivious routing, which packs data into several bins, and routes the bins to a random server through the butterfly network. Both solutions turn any non-oblivious algorithm to the oblivious counterpart. Nevertheless, the communication cost blows up by a constant factor (at least 2) only when the load of the non-oblivious algorithm is balanced, *i.e.*, the number of elements any server received in any round is $O(R/p)$ where $R$ is the total number of elements received of all the $p$ servers. Without load balancing, the communication cost can increase by a factor of up to $p$ in the worst-case scenario. However, existing non-oblivious join algorithms, including the hash join and sort merge join adopted by Spark [58], do not satisfy the constraint. Note that even in the plaintext model where obliviousness is not required, such imbalance happens when the input is skewed, leading to severe performance downgrade. Our join algorithm JODES naturally provides a solution to this issue caused by input skew in the plaintext model, because its performance is independent of the input and hence its skewness.

Opaque [59] proposes an encrypted distributed analytic system based on Spark. Unlike these general solutions, it designs specialized oblivious algorithms for sorting, filter, aggregate, and PK join, but not (general equi-)join. Most of their designs are based on its oblivious sorting, which is implemented based on column sort. SODA [37] considers column sort to be heavy, so it proposes its own oblivious algorithms for filter, aggregate, and join without relying on oblivious sorting, but SODA's join needs to publicize the maximum degrees of the input tables.

A circuit also naturally induces an oblivious algorithm in the distributed setting. To evaluate the circuit, it is necessary to ensure the inputs of each gate lie on the same server. Therefore, it incurs a communication round before each level, hence the number of rounds of the algorithm is linear to the depth of the circuit. However, existing circuits for database joins are all with $\Omega(\log^2 N)$ depth [32, 54] and hence will induce algorithms with polylogarithm number of communication rounds, which severely downgrades the performance due to network latency. Meanwhile, common distributed algorithms introduced above incur only $O(1)$ communication rounds.

*Plaintext distributed join.* Numerous studies have been conducted on join algorithms within the plaintext distributed model, as referenced in [12, 26, 28, 29, 31]. These studies primarily focus on the massively parallel computation model where only the sizes of data received are considered, while disregarding the costs of sending data (including emitting the output) and local computations. However, these algorithms are unsuitable for cloud-based encrypted systems because their local computations, when made oblivious, incur costs that are significantly higher than negligible. Furthermore, the sizes of local joins in their final rounds are data-dependent, which presents challenges for adapting them into oblivious algorithms.

*Join under MPC.* There are several join algorithms [8, 10, 25, 38, 39, 43, 53] under the *secure multi-party computation* (MPC) model, in which several servers jointly compute the join over the secret shared data from the user. The security guarantee of MPC is incomparable to the distributed TEE model: Under MPC, the user does not need to trust any hardware as in TEE, but they believe that the servers will not collude to steal data from the user. In real-world scenarios, the servers in distributed TEE can belong to the same cluster connected by network with low latency and high bandwidth, while servers in MPC are usually from different organizations (*e.g.*, Alibaba, Amazon, and Azure). Regarding efficiency, the speed of join under MPC is usually slower than the one in the standalone TEE setting, which is slower than the distributed TEE setting, All existing join algorithms under MPC incur both computation and communication cost $\Omega(N \log N + M)$ with a considerable hidden constant factor.

## 8 CONCLUSION AND FUTURE WORK

We have proposed JODES, an oblivious algorithm in the distributed setting that is superior to existing works in both theoretical and experimental aspects. Following the idea in [27], one can prove that the communication cost of a perfect load balanced oblivious join (*i.e.*, each server holds $O(M/p)$ of the output tuples of join result) is $\Omega(N + \sqrt{Mp})$. Since the communication costs of existing oblivious join algorithms are all $\Omega(N + M)$, an interesting future research direction is to close the gap, *i.e.*, either proposing an oblivious join algorithm with less cost or providing a stronger lower bound.

## REFERENCES


[1] Foto N. Afrati and Jeffrey D. Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1282–1298. https://doi.org/10.1109/TKDE.2011.47

[2] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (sep 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[3] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An 0(n Log n) Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC '83)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/800061.808726

[4] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1511–1525. https://doi.org/10.1145/3318464.3386141

[5] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure Database-as-a-Service with Cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1033–1036. https://doi.org/10.1145/2463676.2467797

[6] Arvind Arasu and Raghav Kaushik. 2014. Oblivious Query Processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 26–37. https://doi.org/10.5441/002/ICDT.2014.07

[7] 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*. SIAM, 8–14. https://doi.org/10.1137/1.9781611976014.2

[8] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 209–222.

[9] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 307–314. https://doi.org/10.1145/1468075.1468121

[10] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proc. VLDB Endow.* 10, 6 (feb 2017), 673–684. https://doi.org/10.14778/3055330.3055334

[11] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA,) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/2594538.2594558

[12] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication Steps for Parallel Query Processing. *J. ACM* 64, 6, Article 40 (oct 2017), 58 pages. https://doi.org/10.1145/3125644

[13] T-H. Hubert Chan, Kai-Min Chung, Wei-Kai Lin, and Elaine Shi. 2020. MPC for MPC: Secure Computation on a Massively Parallel Computing Architecture. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 151)*, Thomas Vidick (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 75:1–75:52. https://doi.org/10.4230/LIPIcs.ITCS.2020.75

[14] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 803–817. https://doi.org/10.1145/3514221.3517868

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[16] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3–4 (aug 2014), 211–407. https://doi.org/10.1561/0400000042

[17] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3533737.3535098

[18] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (oct 2019), 169–183. https://doi.org/10.14778/3364324.3364331

[19] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* 2, 2–3 (dec 2018), 70–246. https://doi.org/10.1561/3300000019

[20] Facebook. 2022. *Proxygen: Facebook's C++ HTTP Libraries*. https://github.com/facebook/proxygen/releases/tag/v2022.11.14.00 Accessed: 2024-08-12.

[21] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (may 1996), 431–473. https://doi.org/10.1145/233551.233553

[22] 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 Forty-Sixth Annual ACM Symposium on Theory of Computing* (New York, New York) *(STOC '14)*. Association for Computing Machinery, New York, NY, USA, 684–693. https://doi.org/10.1145/2591796.2591830

[23] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, Searching, and Simulation in the MapReduce Framework. In *Algorithms and Computation*, Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 374–383.

[24] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. 2017. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies* 2019 (2017), 370 – 388. https://api.semanticscholar.org/CorpusID:28591687

[25] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiang-Yang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1740–1753. https://doi.org/10.1109/ICDE53745.2022.00176

[26] Xiao Hu. 2021. Cover or Pack: New Upper and Lower Bounds for Massively Parallel Joins. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Virtual Event, China) *(PODS'21)*. Association for Computing Machinery, New York, NY, USA, 181–198. https://doi.org/10.1145/3452021.3458319

[27] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-Optimal Parallel Algorithms for Similarity Joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3034786.3056110

[28] Xiao Hu and Ke Yi. 2019. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Amsterdam, Netherlands) *(PODS '19)*. Association for Computing Machinery, New York, NY, USA, 450–463. https://doi.org/10.1145/3294052.3319698

[29] Xiao Hu and Ke Yi. 2020. Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Portland, OR, USA) *(PODS'20)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3375395.3387657

[30] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society. https://www.ndss-symposium.org/ndss2012/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation

[31] Bas Ketsman and Dan Suciu. 2017. A Worst-Case Optimal Multi-Round Algorithm for Parallel Computation of Conjunctive Queries. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 417–428. https://doi.org/10.1145/3034786.3034788

[32] 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

[33] Tom Leighton. 1984. Tight Bounds on the Complexity of Parallel Sorting. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC '84)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/800057.808667

[34] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[35] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. 2023. Encrypted Databases Made Secure Yet Maintainable. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 117–133. https://www.usenix.org/conference/osdi23/presentation/li-mingyu

[36] Xiang Li, Fabing Li, and Mingyu Gao. 2023. Flare: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. *Proc. VLDB Endow.* 16, 6 (feb 2023), 1439–1452. https://doi.org/10.14778/3583140.3583158

[37] Xiang Li, Nuozhou Sun, Yunqian Luo, and Mingyu Gao. 2023. SODA: A Set of Fast Oblivious Algorithms in Distributed Secure Data Analytics. *Proc. VLDB Endow.* 16, 7 (mar 2023), 1671–1684. https://doi.org/10.14778/3587136.3587142

[38] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX

Association, 1031–1056.

[39] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 1271–1287.

[40] N. Ngai, I. Demertzis, J. Ghareh Chamani, and D. Papadopoulos. 2024. Distributed & Scalable Oblivious Sorting and Shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 156–156. https://doi.org/10.1109/SP54263.2024.00153

[41] 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* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1570–1581. https://doi.org/10.1145/2810103.2813695

[42] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors *(SEC'16)*. USENIX Association, USA, 619–636.

[43] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2129–2146. https://www.usenix.org/conference/usenixsecurity21/presentation/poddar

[44] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 85–100. https://doi.org/10.1145/2043556.2043566

[45] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. 264–278. https://doi.org/10.1109/SP.2018.00025

[46] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proc. VLDB Endow.* 16, 4 (dec 2022), 601–614. https://doi.org/10.14778/3574245.3574248

[47] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2565–2579. https://doi.org/10.1145/3548606.3560603

[48] Amazon Web Services. 2024. *Amazon EMR pricing*. https://aws.amazon.com/emr/pricing/ Accessed: 2024-08-12.

[49] 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. *J. ACM* 65, 4, Article 18 (apr 2018), 26 pages. https://doi.org/10.1145/3177872

[50] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (feb 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[51] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (aug 1990), 103–111. https://doi.org/10.1145/79173.79181

[52] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. 2022. Operon: An Encrypted Database for Ownership-Preserving Data Management. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3332–3345. https://doi.org/10.14778/3554821.3554826

[53] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1969–1981. https://doi.org/10.1145/3448016.3452808

[54] Yilei Wang and Ke Yi. 2022. Query Evaluation by Circuits *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/3517804.3524142

[55] Yilei Wang, Xiangdong Zeng, Sheng Wang, and Feifei Li. 2024. *Jodes: Efficient Oblivious Join in the Distributed Setting*. https://raw.githubusercontent.com/Aqua-Dream/Jodes/main/Jodes_full.pdf

[56] 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 (SP '15)*. IEEE Computer Society, USA, 640–656. https://doi.org/10.1109/SP.2015.45

[57] YSU-Data-Lab. 2024. *TPC-H-Skew*. https://github.com/YSU-Data-Lab/TPC-H-Skew Accessed: 2024-08-12.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, 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 Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 2.

[59] 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 Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 283–298.

[60] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. 2021. Full Encryption: An End to End Encryption Mechanism in GaussDB. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2811–2814. https://doi.org/10.14778/3476311.3476351

[61] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News* 32, 5 (oct 2004), 72–84. https://doi.org/10.1145/1037947.1024403