Secure Sampling for Approximate Multi-party Query Processing

QIYAO LUO, Hong Kong University of Science and Technology, Hong Kong SAR, China
YILEI WANG, Alibaba Group, Hangzhou, China
KE YI, Hong Kong University of Science and Technology, Hong Kong SAR, China
SHENG WANG, Alibaba Group, Hangzhou, China
FEIFEI LI, Alibaba Group, Hangzhou, China

We study the problem of random sampling in the secure multi-party computation (MPC) model. In MPC, taking a sample securely must have a cost $\Omega(n)$ irrespective to the sample size s. This is in stark contrast with the plaintext setting, where a sample can be taken in $O(s)$ time trivially. Thus, the goal of approximate query processing (AQP) with sublinear costs seems unachievable under MPC. To get around this inherent barrier, in this paper we take a two-stage approach: In the offline stage, we generate a batch of n/s samples with $\tilde{O}(n)$ total cost, which can then be consumed to answer queries as they arrive online. Such an approach allows us to achieve an $\tilde{O}(s)$ amortized cost per query, similar to the plaintext setting. Based on our secure batch sampling algorithms, we build MASQUE, an MPC-AQP system that achieves sublinear online query costs by running an MPC protocol to evaluate the queries on pre-generated samples. MASQUE achieves the strong security guarantee of the MPC model, i.e., nothing is revealed beyond the query result, which itself can be further protected by (amplified) differential privacy.

CCS Concepts: • **Security and privacy → Database and storage security**; • **Information systems → Data management systems**; • **Mathematics of computing → Probability and statistics**.

Additional Key Words and Phrases: Secure multi-party computation; Sampling; Approximate query processing

ACM Reference Format:
Qiyao Luo, Yilei Wang, Ke Yi, Sheng Wang, and Feifei Li. 2023. Secure Sampling for Approximate Multi-party Query Processing. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 219 (September 2023), 27 pages. https://doi.org/10.1145/3617339

1 INTRODUCTION

Data analysis often needs to be conducted across multiple organizations. Meanwhile, as data gets increasingly valuable, there is also a growing need to protect its security and privacy.

Example 1.1. An insurance company wishes to estimate the budget it should prepare for certain types of diseases in 2023. This information could be obtained by running a query on the joint data from the insurance company and the hospitals. For example, several hospitals jointly hold R(person, year, disease) and the insurance company has relation S(person, cost, rate). A simple SQL query on these relations can be the following:

Authors' addresses: Qiyao Luo, qluoak@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong SAR, China; Yilei Wang, fengmi.wyl@alibaba-inc.com, Alibaba Group, Hangzhou, China; Ke Yi, yike@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong SAR, China; Sheng Wang, sh.wang@alibaba-inc.com, Alibaba Group, Hangzhou, China; Feifei Li, lifeifei@alibaba-inc.com, Alibaba Group, Hangzhou, China.

2836-6573/2023/9-ART219 $15.00
https://doi.org/10.1145/3617339

```
SELECT disease, SUM(cost * rate)
FROM R, S
WHERE R.person = S.person AND R.year = 2023
GROUP BY disease
```

However, the medical records at the hospitals, as well as the insurance policies, contain sensitive personal information, and should not be revealed to other parties. A query processing system under the secure multi-party computation (MPC) model thus meets this requirement.

MPC enables multiple parties to jointly compute a function without disclosing their private input. Unfortunately, the current MPC-SQL systems are 1000+ times slower than on plaintext [7, 44]. For instance, it takes SMCQL [7] 1000 seconds to evaluate a query over only 200 tuples. Most existing works improve the performance by weakening the security guarantee or restricting to a small class of queries. Conclave [52] relies on a trusted third party; Scape [27] reveals the intermediate join sizes; Shrinkwrap [8] protects the intermediate join sizes by differential privacy but at a high cost of at least $\Omega(n^2)$; Secure Yannakakis [53] strictly follows the MPC security requirement, but supports only limited types of queries. In this paper, we aim at designing an MPC-SQL system that does not compromise on security or generality; the sacrifice we make is that only approximate query results are returned.

In fact, even for query processing over plaintext, where performance is not a big issue unless running on very large datasets, *approximate query processing (AQP)* has already been extensively studied [1, 15, 29, 37, 38, 42]. Thus, AQP is a more valuable approach for MPC even on not-so-large datasets. Among the many AQP techniques, random sampling is the most appealing due to its versatility in handling a large class of queries. Meanwhile, there is a rich body of work from the statistics literature that can be used to derive statistically sound interpretations of the query results evaluated on the sample. A general rule of thumb is that the (normalized) sampling error is roughly proportional to $1/\sqrt{s}$ for sample size s. This means that the sample size, hence the query processing cost, can be sublinear or even independent to the database size n, which is especially appreciated when facing large data sets.

Due to the high overhead of MPC protocols, sampling based AQP techniques are in an even stronger demand than on plaintext. In Example 1.1, since the insurance company does not necessarily require an exact result, the protocol can work on a sample so as to reduce the query evaluation time.

Indeed, this motivation is well articulated in SAQE [9], the first MPC-AQP system. Note that the sample must be taken securely, i.e., no one should know which elements are taken (or not) into the sample; otherwise, a curious onlooker may be able to deduce unauthorized information by linking this knowledge with the query result [9]. However, the MPC sampling protocol of [9] has a cost (both running time and communication) of $O(n \log n)$, no matter how small the sample size s is. This may not be faster than exactly computing the query result without sampling, which has $O(n)$ cost for many queries but with a larger hidden constant.

Unfortunately, there is an inherent $\Omega(n)$ lower bound for secure sampling (so the algorithm of [9] cannot be improved by more than a logarithmic factor): If a record is not touched, then it reveals that it must not be in the sample. To get around this barrier, in this paper, we take a two-stage approach: In the offline stage (before any query is given), we solve the *batch sampling* problem, i.e., generating n/s samples of size s each, with a total cost of $\tilde{O}(n)$[1]. Thus, the amortized cost per sample is $\tilde{O}(s)$. Then during the online stage, the samples are consumed as queries arrive, such that the online query processing cost is $\tilde{O}(s)$, similar to the plaintext setting (but with a larger hidden

[1]The $\tilde{O}$ notation suppresses polylogarithmic factors.

constant, as we need to run an MPC protocol to evaluate the query on the sample). When samples are depleted, we re-run the batch sampling to obtain a fresh set of samples. We also support the case when s is given at query time instead of during the offline stage, with a polylogarithmic factor increase in the offline storage and computation. This idea will be introduced in Section 3.3.

This two-stage approach considerably reduces the response time of queries, as the heavy computation parts have been moved to the offline stage. In fact, most MPC systems already adopt a two-stage design: Oblivious transfer (OT) extension [32] generates a small number of base OTs offline, which can then be used to extend to a large number of OTs online; Beaver triples [10] are generated in an offline stage to accelerate multiplications in the online stage; when data is stored in multiple tables, it is often denormalized using MPC joins [7, 27, 40, 44, 54] in an offline stage, i.e., the fact table is joined with all the dimension tables to form a flat table. In some sense, our proposal is to decouple the sampling process in such a manner as well, which is necessary to break the $\Omega(n)$ lower bound.

1.1 Contributions

Specifically, we make the following contributions in this paper:

(1) We formally introduce the batch sampling problem under MPC, as a necessary way to achieving an $\tilde{O}(s)$ cost amortized per sample.
(2) We describe a suite of MPC batch sampling algorithms including shuffle sampling, sampling with replacement (WR sampling), sampling without replacement (WoR sampling), and stratified sampling. We discuss their pros and cons in terms of cost, independence, sampling error, and (differential) privacy amplification. The algorithm for WoR sampling under MPC is particularly nontrivial. We model the problem as a graph, design a circuit for pointer jumping, and then construct a WoR sampling circuit based on it. We also generalize our WoR sampling algorithm to support stratified sampling.
(3) Based on our sampling algorithms, we build MASQUE (Multi-party Approximate Secure QUEry processing), an MPC-AQP system that utilizes two semi-honest non-colluding servers with any number of data owners. During the offline stage, the data owners first secret-share their data to the two servers. MASQUE then denormalizes the data using secure joins and generates a batch of samples. During the online stage, one sample is consumed for each query received. For all-around privacy protection, MASQUE also optionally injects differential privacy (DP) noise to the query result before returning it to the designated receiver. We empirically compare MASQUE with SAQE [9], the previous MPC-AQP system, and SMCQL [7] and SecYan [53], the state-of-the-art exact MPC query processing engines. Experimental results show that MASQUE can reduce the online latency of MPC query processing significantly.

1.2 Related Work

Sampling based AQP techniques have been extensively studied over plaintext data and many AQP systems exist [1, 15, 29, 31, 37, 42]. Some systems take the sample online after a query arrives [31, 37], while others also take a two-stage approach, where samples are pre-generated during an offline stage [1, 29, 42], similar to what we do in this paper. Since an online sample can be taken in $O(s)$ time over plaintext, the two-stage approach offers limited improvement, and which one performs better delicately depends on the query and data characteristics [15]. This is the major difference between MPC and the plaintext setting, as taking an online sample under MPC requires $\Omega(n)$ cost, so the two-stage approach has a significant advantage in this case.

Secure multi-party computation (MPC), which enables multiple parties to jointly compute a function without disclosing any participant's private input except the function output, was first conceptualized by Yao in his pioneering paper [56]. Generic protocols, such as *Yao's garbled circuits* [57], *GMW* [24], and *BGW* [11], are all based on expressing the computation as an arithmetic or Boolean circuit. For many problems, such as compaction and expansion (see Section 2.4), circuit-based protocols are still the best solution, although for certain problems, such as sorting [3, 4, 16, 26], permutation [16, 41], and private set intersection (PSI) [33, 40, 43], custom protocols have been developed with lower costs under some particular MPC models.

Several exact MPC query processing engines have been developed [7, 40, 44, 53]. All of them have a query processing cost of at least $\Omega(n)$ with a large hidden constant due to the overhead of the MPC protocols mentioned above. To lower the query processing cost at the expense of accuracy, SAQE [9] takes a random sample under MPC and then evaluates the query on the sample. However, as mentioned, the sampling step still has $\tilde{O}(n)$ cost. In the trusted execution environment (TEE) model, a batch sampling algorithm has been proposed in [49] with an amortized cost of $\tilde{O}(s)$ per sample, but their algorithm does not work in the MPC model. There are also some works that generate secure sample indices with a specific distribution [14, 17, 45, 46] in MPC. They focus on sampling from a weighted (non-uniform) distribution and output sample indices in plaintext. However, all of them have $\Omega(n)$ sampling cost.

2 PRELIMINARIES

2.1 MPC Basics

In the MPC model, several parties jointly compute a given function. An MPC protocol ensures that the involved parties only learn the output, while keeping their inputs secret.

Complexity measures. The complexity of an MPC protocol is measured by computation time (the total computation time of all the parties during the protocol), communication cost (the total size of messages sent during the protocol), and the number of communication rounds. In the complexity expressions, we suppress the common factors such as the number of bits used to represent each tuple and the computational security parameter.

Adversary. The security of an MPC protocol is measured by its ability to defend against some adversary. A *semi-honest* adversary can see the transcripts (i.e., all the messages sent and received during the protocol) of some parties, which we call *corrupted parties*, and will try to infer information of other parties based on them. A *malicious* adversary in addition has the ability to change the transcript (i.e., deviate from the protocol) arbitrarily during the protocol, for the purpose of stealing information of other parties.

An adversary can also be either computationally bounded or unbounded. For the former, the adversary has polynomial time to break the protocol, while there is no limit for the latter. We only consider a computationally bounded adversary in this paper.

Secret-sharing. During an MPC computation, intermediate and final results are often stored in a *secret-shared* form. We adopt the well-known Boolean sharing in the two-party model, where a value v with ℓ bits is split as $v = [\![v]\!]_0 \oplus [\![v]\!]_1$. The i-th party holds $[\![v]\!]_i, i \in \{0, 1\}$, which consists of ℓ random bits, and $\oplus$ represents for the logic xor operation.

Circuit. A circuit is a directed acyclic graph (DAG), where each gate (node) is either an input (node without incoming edges), an operator, or an output (node without outgoing edges). Common operators include addition, subtraction, multiplication (i.e., arithmetic circuit), logic and, logic xor (Boolean circuit). When a circuit is evaluated, the inputs go through the wires (edges), transformed

by the operators, and are finally outputted. Under MPC, the elements on all the gates are in secret-shared form, so the evaluation process is secure.

Circuits are a powerful tool for MPC, as they separate algorithm (circuit) design from the actual implementation (the exact form of secret-sharing and which MPC protocol is to be used to evaluate the gates). One algorithm can thus be used for any MPC variant, no matter how the data are initially distributed, how many parties there are, how many of them can be corrupted, whether the adversary is semi-honest or malicious, etc. All these MPC models have been studied in depth in the security literature, with secret-sharing methods and MPC protocols carefully designed for each. Thus, one may choose the right implementation for a particular application scenario to plug into the circuit. For example, Yao's garbled circuit is suitable for the semi-honest two party model, while the SPDZ framework [30] is designed for the malicious multi-party setting. Both the computation time and communication cost are proportional to the size of the circuit (i.e., total number of gates), while the number of communication rounds is either constant or linear to the depth of the circuit (i.e., the diameter of the DAG), depending on the protocol.

Although it is possible to automatically transform any algorithm into a circuit, the efficiency blow up could be very large [39]. Finding a circuit with size complexity the same as the algorithm is non-trivial or even impossible, and the sampling problem is such an example. In this paper, we propose the batch sampling problem to break this limitation, and design circuits for it. All the circuits we design have size matching their RAM counterparts, despite a polylogarithm factor. The depths of these circuits are also polylogarithm. These circuits directly imply an efficient MPC protocol.

Using a circuit may not provide the most efficient solution for a particular problem under a particular MPC model. The most notable example is sorting, where the best circuit has $O(n \log^2 n)$ size[2], but under the two-server model, which is what MASQUE is built upon and more precisely defined in Section 6, there are $O(n \log n)$-cost sorting protocols [26].

Thus, for the algorithmic results in this paper (i.e., the batch sampling problem), we design a circuit for its generality and conceptual simplicity. When describing MASQUE in Section 6, we specialize into the two-server model with more efficient custom protocols for certain parts of the circuit (in particular, sorting). Furthermore, we use ABY [18], a well-known 2PC framework that can evaluate garbled circuits with Boolean-shared inputs and produce Boolean-shared outputs.

Security definition. We follow the standard security definition in MPC, which is based on the *real-ideal paradigm*. In the "ideal world", there exists a trusted third party which collects all the data from the parties, computes the function locally, and sends the output to the parties. The *view* of a party (in either the ideal or real world) consists of its private input, and messages it sent and received during the protocol (including the output). The view of an adversary includes the views of all the corrupted parties. A real-world protocol is secure if for any input and any adversary $\mathcal{A}$, there exists a simulator in the ideal world which can produce a view of $\mathcal{A}$ that is computationally indistinguishable from $\mathcal{A}$'s real-world view (i.e., $\mathcal{A}$ cannot tell whether the views are from the real-world or the ideal-world with more than negligible probability). Note that for functions that involve internal randomness (e.g., compute a function from a random sample), the randomness is not in the view of the adversary.

The security of an MPC protocol is specified by two parameters: the computational security parameter κ and the statistical security parameter σ. The former indicates the difficulty for a computationally bounded adversary to break the protocol, which decides the encryption key length; the latter corresponds to a failure probability of $2^{-\sigma}$ (either security is broken or the algorithm fails to return the correct result) even assuming an unbounded adversary. For the circuit itself, only σ

[2]Theoretically, $O(n \log n)$-size sorting circuits exist, but the hidden constant is impractically large.

Methods	Independence	Sampling error	Privacy amplification	Circuit depth	Circuit size
Shuffle sampling	No	$O\left(\frac{1}{\sqrt{s}} \cdot \sqrt{\frac{n-s}{n}}\right)$	$(\varepsilon, 0)^*$	$O(\log^2 n)$	$O(n \log^2 n)$
WR sampling	Yes	$O\left(\frac{1}{\sqrt{s}}\right)$	$\left(\frac{s}{n} \cdot \varepsilon, O\left(\frac{s^2}{n^2} \cdot \varepsilon\right)\right)^\star$	$O(\log^2 n)$	$O(n \log^2 n)$
WoR sampling	Yes	$O\left(\frac{1}{\sqrt{s}} \cdot \sqrt{\frac{n-s}{n}}\right)$	$\left(\frac{s}{n} \cdot \varepsilon, 0\right)^\dagger$	$O(\log^2 n \log \sigma)$	$O(n \log^2 n \log \sigma)$
Stratified sampling	Yes	$O\left(\frac{1}{\sqrt{k_i}} \cdot \sqrt{\frac{d_i - k_i}{d_i}}\right)^\ddagger$	$\left(\frac{k_i}{d_i} \cdot \varepsilon, 0\right)$	$O(\log^2 n \log \sigma)$	$O(n \log^2 n \log \sigma)$

*Shuffle sampling provides no privacy amplification;

⋆WR sampling has $\varepsilon' = \log((1 - (1 - \frac{1}{n})^s) \cdot (e^\varepsilon - 1) + 1) \approx s/n \cdot \varepsilon$ and $\delta' \le \sum_{k=1}^{s} \binom{s}{2} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{s-k} \left(\frac{\varepsilon}{2} - \frac{\varepsilon}{2k}\right) = O\left(\frac{s^2}{n^2} \cdot \varepsilon\right)$;

†WoR sampling has $\varepsilon' = \log(s/n \cdot (e^\varepsilon - 1) + 1) \approx s/n \cdot \varepsilon$ and keeps pure-DP;

‡Each stratum, which takes k_i WoR samples from d_i data, has the same sampling error and privacy amplification as WoR sampling.

Table 1. Comparisons of different sampling methods

matters, while both κ and σ should be specified when implementing the gates of a circuit using a particular MPC protocol. In practice, the value of σ is often taken to be 40 or 80; for the asymptotic results in this paper, we assume $\sigma = \Omega(\log n)$.

2.2 Sampling Basics

Sampling is the most simple and popular method for approximate computation. For an input sequence $\mathcal{X} = (x_1, \ldots, x_n)$, a sampling method returns a multiset S such that any $x \in S$ is randomly taken from $\mathcal{X}$. Different sampling probability or correlation yields different strategies. We introduce the common strategies in this section.

Sampling with replacement (WR sampling): Each $x \in S$ is uniformly randomly chosen from $\mathcal{X}$ independently. In this case, we also call S a *WR sample*. Note that even with $s = n$, not every element is ensured to appear in the WR sample, because the same element may appear multiple times. Thus in this case, the sampling error is still non-zero.

The WR sampling algorithm in plaintext is trivial: For each step, randomly draw an integer i uniformly from $\{1, \ldots, n\}$ and take x_i to the sample, and repeat this step for s times. This algorithm runs in $O(s)$ time. However, under MPC model this complexity is not achievable.

Sampling without replacement (WoR sampling): S is a subset of $\mathcal{X}$ with size s, i.e., elements in S are all different. Each subset of s elements of $\mathcal{X}$ are taken as the sample with equal probability. A WoR sample usually has better quality than a WR sample (with equal sample size) especially when s is close to n. Specifically, when $s = n$, $S = \mathcal{X}$ and there is no sampling error.

The standard WoR sampling algorithm is straightforward: We simply sample an element uniformly at random, remove it from $\mathcal{X}$, and repeat the process s times. However, this algorithm is inherently sequential and relies on the power of the RAM. Thus, it cannot be expressed by a circuit. Instead, we will use another sampling algorithm, known as Floyd's algorithm [12], shown in Algorithm 1. For convenience, the described algorithm only returns the indices of the sample. Still, Floyd's algorithm is a sequential RAM algorithm. It cannot be directly implemented as a circuit, as circuit does not support set operations (i.e., line 4, 5, and 7) in constant time. However, we will show later how to model it as a graph problem, which then can be solved by a circuit.

Stratified sampling: All sampling methods above are uniform in the sense that every element is sampled with equal probability. Stratified sampling is a variant of WoR sampling, and is a commonly used non-uniform sampling method, especially useful for group-by queries to ensure that small groups are also properly represented in the sample. Here, elements in $\mathcal{X}$ are divided into g strata where the j-th stratum has size d_j. The sample S is the union of g WoR independent subsamples, the

j-th subsample of which has size k_j and is drawn from the j-th stratum, where k_j is a non-negative integer decided by a given stratified sampling policy, which will be introduced in Section 5. WoR sampling can be viewed as a special stratified sampling with $g = 1$. In Example 1.1, the groups for stratified sampling could be the classes of disease types.

If each stratum appears consecutively in $\mathcal{X}$, it is direct to apply WoR sampling algorithm for each stratum to get an algorithm for stratified sampling in plaintext, which also has cost $O(s)$. However, the problem becomes nontrivial if all d_j, k_j, and even g are private.

Algorithm 1: Floyd's sampling algorithm

Input: Data size n; sample size s
Output: A set S of s indices taken from $\{1, 2, \cdots, n\}$ uniformly without replacement

```
S ← ∅;
for b ← n − s + 1 to n do
    a ← a uniform random index in {1, 2, ··· , b};
    if a ∈ S then
        Insert b to S;
    else
        Insert a to S;
return S
```

2.3 Differential Privacy & Privacy Amplification

While MPC ensures that the transcript of the protocol does not leak any private information, the output of the public function is still revealed, which may contain private information. *Differential privacy (DP)* [22] is possibly the most popular solution to address this issue. An algorithm $\mathcal{M}$ is (ε, δ)-differential private if for any two neighbouring inputs $\mathbf{I} \sim \mathbf{I}'$ and any set of outputs Y,

$$\Pr[\mathcal{M}(\mathbf{I}) \in Y] \leq e^{\varepsilon} \cdot \Pr[\mathcal{M}(\mathbf{I}') \in Y] + \delta.$$

Here ε and δ are the privacy parameters, where smaller values correspond to stronger privacy guarantees but lead to larger errors. In particular, it is also called *pure-DP* if $\delta = 0$. The Laplace mechanism is a standard approach to inject DP noises, which can be easily expressed by a circuit and then implemented in MPC [9].

Sampling is a useful tool in differential privacy, as a DP mechanism running on a random sample provides higher privacy guarantees than when on the entire input. Formally, if $\mathcal{M}$ is an (ε, δ)-DP mechanism and $\mathcal{S}$ is a sampling mechanism, then $\mathcal{M} \circ \mathcal{S}$ satisfies $(\varepsilon', h(\delta))$-DP for some $\varepsilon' \leq \varepsilon$ and some function h. This is known as *privacy amplification* because the new mechanism has better (i.e., smaller) ε. Table 1 summaries the privacy amplification bounds for pure-DP for different sampling methods. Shuffle sampling, which will be introduced in Section 3.1, does not have privacy amplification [49], and the privacy amplification of WR sampling does not keep pure-DP [5]. Only WoR sampling (and its variant, stratified sampling) provides privacy amplification and keeps pure-DP at the same time [5].

2.4 Basic Circuits

We introduce some basic circuits, which will be used to construct our batch sampling circuits later.

Uniform Random Number Generator. A circuit has a fixed structure. To support random sampling, some input wires will need to be fed with random numbers. A random bit under any secret-sharing method can be easily generated [9, 13]. From these random bits, one can generate a uniformly random (secret-shared) number from $[x] = \{1, \ldots, x\}$, where x is also given in a secret-sharing form. The idea is to take σ random bits, which combined together form a uniform random number $y \in \{0, \ldots, 2^\sigma - 1\}$. Then we return $(y \bmod x) + 1$. Generating y in a range of 2^σ ensures that the generated random number is $2^{-\sigma}$-close to uniform, even if x does not divide 2^σ. We use $\mathrm{URNG}(x)$ to denote such a random number generator.

Sorting. Sorting circuits, a.k.a. sorting networks, have been studied extensively. The AKS network [2] achieves the asymptotically optimal $O(n \log n)$ size and $O(\log n)$ depth, but with a huge hidden constant. In practice, the Bitonic sorter [6] is widely used, which has $O(n \log^2 n)$ size and $O(\log^2 n)$ depth. In this paper, we use the bitonic sorter in our circuit, denoted by `sort`, although it will be replaced by a more efficient sorting algorithm in the two-server MPC model [26], which MASQUE adopts.

Prefix-sum. The *prefix-sum* operation takes a sequence $X = (x_1, \ldots, x_n)$ and a binary associative operator $\oplus$, and outputs the sequence $(x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n)$. We call the output the `prefix-`$\oplus$ of X. If $\oplus$ can be evaluated by a constant-size circuit, there is a `prefix-`$\oplus$ circuit with $O(n)$ size and $O(\log n)$ depth [36].

One useful generalization of prefix-sum is *segmented prefix-sum*. It takes two sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ as input, where equal elements in A must appear consecutively and form a segment. The outputs are the prefix-sums of each segment in B. For example, if $A = (2, 2, 4, 1, 1)$, then the output is $(b_1, b_1 + b_2, b_3, b_4, b_4 + b_5)$. We call the output the `prefix-`$\oplus$ of B segmented by A. The segmented prefix-sum problem can also be solved by a circuit of $O(n)$ size and $O(\log n)$ depth [54].

Compaction. The input of the *compaction* operation consists of two sequences $X = (x_1, \ldots, x_n)$ and $T = (t_1, \ldots, t_n)$, where each $t_i \in \{0, 1\}$. Each x_i with $t_i = 1$ is said to be *marked*. The compaction of X on T returns a permutation of X such that all marked elements appear before the unmarked elements. We require the compaction to be *order-preserving*, i.e., the relative ordering of the marked elements must be preserved. We denote this operation as "`compact` X by T". It can be solved by a circuit of $O(n \log n)$ size and $O(\log n)$ depth [48]. A special case of compaction is when X contains dummy elements (denoted as $\perp$) while there is no T. In this case, the compaction moves all non-dummy elements to the front, still in an order-preserving fashion.

Expansion. The *expansion* operation takes two sequences $X = (x_1, \ldots, x_n)$ and $D = (d_1, \ldots, d_n)$, where each d_i is a positive integer that indicates the number of repetitions that x_i should appear in the output. The output is a length-m sequence for $m = \sum_{i=1}^{n} d_i$:

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

We denote this operation as "`expand` X by D".

Expansion circuits have not been explicitly described in the literature. Nevertheless, it is not hard to modify the algorithm of [35], which is designed under the oblivious RAM model, into an expansion circuit of $O(m \log m)$ size and $O(\log m)$ depth. The idea works as follow: First calculate the prefix-sum of D, which indicates the first location where each distinct x_i appears; then put each x_i to its destination using the butterfly-like network [25] in a reverse order; and finally copy each x_i to fill its following empty locations by another prefix-sum circuit. The concrete circuit is

described in Algorithm 2, in which the binary operator $\oplus$ is defined as

$$x \oplus y = \begin{cases} x, & \text{if } y = \perp; \\ y, & \text{otherwise.} \end{cases}$$

Note that although we present the circuit using pseudocode instead of drawing it out, it should be clear that it yields a circuit. In particular, the **if-then-else** can be done by a series of comparison gates and multiplexers. The same holds for all other circuits presented later in this paper.

Algorithm 2: Expansion circuit

Input: $X = (x_1, \ldots, x_n)$; $D = (d_1, \ldots, d_n)$
Output: $Y = (y_1, \ldots, y_m)$

$(e_1, \ldots, e_n) \leftarrow$ `prefix-+` of $(0, d_1 - 1, d_2 - 1, \ldots, d_{n-1} - 1)$;
for $j \leftarrow 1$ **to** m **do in parallel**
  if $j \le n$ **then**
    $(a_j^{(0)}, b_j^{(0)}) \leftarrow (x_j, e_j)$;
  else
    $(a_j^{(0)}, b_j^{(0)}) \leftarrow (\perp, 0)$;
for $\ell \leftarrow 1$ **to** $\lceil \log m \rceil$ **do**
  $s \leftarrow 2^{\lceil \log m \rceil - \ell}$;
  for $j \leftarrow 1$ **to** m **do in parallel**
    if $b_j^{(\ell-1)} < s$ **and** $a_j^{(\ell-1)} \neq \perp$ **then**
      $(a_j^{(\ell)}, b_j^{(\ell)}) \leftarrow (a_j^{(\ell-1)}, b_j^{(\ell-1)})$;
    else if $j > s$ **and** $b_{j-s}^{(\ell-1)} \ge s$ **then**
      $(a_j^{(\ell)}, b_j^{(\ell)}) \leftarrow (a_{j-s}^{(\ell-1)}, b_{j-s}^{(\ell-1)} - s)$;
    else
      $(a_j^{(\ell)}, b_j^{(\ell)}) \leftarrow (\perp, 0)$;
$Y \leftarrow$ `prefix-`$\oplus$ of $(a_1^{(\ell)}, \ldots, a_m^{(\ell)})$;
return Y

Primary Key Join. Join, or natural join, is a basic operation in databases. Let $R(A, B)$ and $S(B, C)$ be two relations, each of which has n tuples. Their join result, $R \bowtie S$, is the combination of tuples of R and S that have the same value for their common attribute B, i.e.,

$$R \bowtie S = \{(a, b, c) \mid (a, b) \in R \text{ and } (b, c) \in S\}.$$

Doing the join using a circuit must prepare for the worst case where the output size can be n^2. We avoid such a large circuit by only using primary key joins, where B is the *primary key* of R. Such a join has output size at most n, and there is a circuit of size $O(n \log^2 n)$ and depth $O(\log^2 n)$ for computing it (see Section 5.3 of [54]). This circuit has an output size of n, which is the maximum possible join size, although some output elements may be $\perp$.

3 BATCH SAMPLING

We propose *batch sampling*, in which both the input $\mathcal{X}$ and output $\mathcal{S}$ are a sequence of n elements, such that each $S_i = \mathcal{S}[(i-1)s + 1, \ldots, is]$, $i = 1, \ldots, n/s$, is a random sample taken from $\mathcal{X}$. When

each S_i is a WR sample, WoR sample, or stratified sample, and all S_i are independent, we call it batch WR sampling, batch WoR sampling, or batch stratified sampling, respectively, and their algorithms in plaintext can be easily obtained by repeatedly generating the corresponding sample for n/s times independently, which have $O(n)$ cost. The main algorithmic results of this paper are $\tilde{O}(n)$-size circuits for these sampling methods. See Table 1 for a summary.

3.1 Shuffle Sampling

Shuffle sampling differs from batch WR/WoR sampling that all S_i are not independent. For shuffle sampling, $\mathcal{S}$ is simply a random permutation of $\mathcal{X}$, i.e., each S_i is a WoR sample and the set of $\mathcal{X}$ is a union of all S_i. In plaintext, a random permutation on $\mathcal{X}$ can be obtained by applying the Fisher-Yates shuffle algorithm [23]. Due to this simplicity, shuffle sampling is widely used in stochastic gradient descent. However, since the samples are not independent, it causes issues in query estimation, fairness, and representativeness in an AQP system [50].

Under MPC, the standard approach to implement shuffle sampling is by first assigning each element a sufficiently long random key (such that any pair of keys are different) and sorting the elements by the keys. More precisely, $(\sigma + 2\log n)$-bit random keys for n elements ensure that repetitive keys appear with probability at most $2^{-\sigma}$, by the similar analysis to the birthday problem. Thus, the shuffle sampling circuit has the same size and depth as that of sorting.

3.2 The Batch WR Sampling Circuit

We introduce our construction to the batch WR sampling circuit. It also serves as an example to illustrate how to break the $\Omega(n)$ lower bound by batch sampling. The key observation is that, although taking an element from $\mathcal{X}$ with a secret random index requires $\Omega(n)$ time, generating such a random index can be done in $O(1)$ time by URNG. Therefore, we first generate *eid*, the random indices of elements in all the n/s samples, along with the corresponding *sid*, indicating which sample this element belongs to. Then we use primary key join to connect the indices to their corresponding elements. Finally sort the elements by *sid* so elements in the same sample appear consecutively. See Algorithm 3 for the pseudocode.

Note that the idea also works for batch WoR sampling or batch stratified sampling: Given a circuit that generates the s indices of elements for each sample, we apply the circuit n/s times independently to get n indices of elements for the batch of samples, and then map the indices to the elements by a primary key join. Therefore, we will discuss how to generate the indices of a WoR sample and a stratified sample in Section 4 and 5 respectively.

Algorithm 3: Batch WR sampling circuit

Input: $\mathcal{X} = (x_1, \ldots, x_n)$
Output: $\mathcal{S} = \{S_i\}_{i=1}^{n/s}$

Initialize $R_1(X, eid)$ with size n;
Initialize $R_2(eid, sid)$ with size n;
for $i \leftarrow 1$ **to** n **do in parallel**
    $R_1[i] \leftarrow (x_i, i)$;
    $R_2[i] \leftarrow (\text{URNG}(n), \lceil i/s \rceil)$;
Compute $T \leftarrow R_1 \bowtie R_2$, where *eid* is the primary key of R_1;
`Sort` T on $T.sid$;
return $T.X$

3.3 Online Sample Size

The batch sampling, by definition, requires the sample size s given in advance and all samples in the batch have the same size. In many applications, the user may desire a different sample size for each query, especially for interactive data analytics. One may start with some rough estimates (small sample sizes suffice) and then drill down for more accurate answers.

For batch WR sampling or shuffle sampling, it is straightforward to return the first s' elements of $\mathcal{S}$ to the query, if it specifies sample size s'. This solution does not work for WoR sampling or stratified sampling, though. To support online sample sizes, we can prepare $\lceil \log n \rceil$ batches of samples in the offline stage, where the sample sizes are $s = 2, 4, 8, \cdots, 2^{\lceil \log n \rceil - 1}, n$. That is, the i-th batch contains $n/2^i$ samples of size 2^i. When the user wants a sample of size s', we take a sample from the $\ell = \lceil \log s' \rceil$-th batch, which has size 2^ℓ and is at most twice the required[3] sample size s'. This increases the initial offline cost by a $\log n$ factor. However, it is important to note that the amortized sampling cost does not increase, because after a batch is depleted, we only need to replenish that batch.

3.4 Private Data Size

The default constructions of the circuits of batch sampling in this paper assume that the input $\mathcal{X}$ contains no dummy elements. In many MPC applications, dummy elements are often padded to hide the number of real elements. When taking samples from a dataset with dummies, we should guarantee that we only sample from the real elements, otherwise the quality of the sample is not ensured.

Let $\tilde{n} \leq n$ be the number of real elements, which is private. This means that the circuit structure cannot depend on $\tilde{n}$. Shuffle sampling does not easily extend to this case.[4] Our circuits for batch WR sampling, WoR sampling, and stratified sampling, can easily support this scenario with little overhead, which is an extra compaction that move dummy elements to the end of $\mathcal{X}$, so that the first $\tilde{n}$ elements are real. Then, for our WR sampling circuit, we simply generate random indices ($S.eid$) in $[\tilde{n}]$ instead of $[n]$. This idea also works similarly in the WoR sampling circuit or stratified sampling circuit. We ignore the details in this paper.

4 WOR SAMPLING

In this section, we discuss how to design a circuit with size $\tilde{O}(s)$ and depth $\tilde{O}(1)$ that generates indices for a WoR sample. As mentioned in Section 3.2, this yields a batch WoR sampling circuit with size $\tilde{O}(n)$ and depth $\tilde{O}(1)$. Compared with WR sampling, the major difficulty is ensuring the indices of a WoR sample to be distinct. All existing algorithms in plaintext will lose their high efficiency under MPC model. For example, simply simulating the Floyd's algorithm (Algorithm 1) with a circuit would have $O(s^2)$ size per sample, as it needs a membership test $b \in S$ in line 4. A naive circuit for this still requires $O(s)$ size. Besides, the algorithm has s sequential steps; we must parallelize it in order to reduce the circuit depth.

Below we show how to transform Floyd's algorithm into a circuit of size $\tilde{O}(s)$ and depth $\tilde{O}(1)$. Our idea is to model the execution of Floyd's algorithm with a directed graph $G = (V, E)$. The nodes $V = [n]$ correspond to the indices to be sampled, the edges are $E = \{(a_i, b_i)\}_{i=1}^{s}$ where $b_i = n - s + i$ and each a_i is taken from URNG(b_i), and a_i and b_i correspond to the values of a and b respectively in the i-th iteration of the loop. Note that $a_i \leq b_i$ for all i. We allow self-loops, i.e., $a_i = b_i$. Let S be

[3]We could further sub-sample to obtain the required sample size exactly, but this is often unnecessary: Returning a slightly larger sample only makes the query more accurate, while incurring the same query processing cost $O(s)$.
[4]If we only return the random permutation over the $\tilde{n}$ real elements, the output size would reveal $\tilde{n}$.

the output of Algorithm 1, then $S \subseteq V$ forms a size-s WoR sample of $[n]$. We then discuss how to find S using this graph.

4.1 The Reduced Graph

First we introduce the *reduced graph* $G'(V', E')$ of G, where V' is the non-trivial nodes of V, i.e.,

$$V' = \{\exists b : (a, b) \in E \mid a \in V\} \cup \{\exists a : (a, b) \in E \mid b \in V\}.$$

The edge set E' is initially $\emptyset$. Define $V_a = \{b \in V \mid (a, b) \in E\}$. For any a where $V_a \neq \emptyset$, assume the elements in V_a are $v_1, \ldots, v_k$ in the order $v_1 < v_2 < \cdots < v_k$, then we insert the set of edges $\{(a, v_1), (v_2, v_2), \ldots, (v_k, v_k)\}$ to E'. The circuit for computing the edges of the reduced graph is described in Algorithm 4. See Figure 1 for an example.

Algorithm 4: Construct reduced graph

Input: Edges $E = \{(a_i, b_i)\}_{i=1}^{s}$;
Output: Edges of reduced graph G'

```
1 Initialize relation R(A, B) with E;
2 Sort R by (A, B);
3 for i ← 2 to s do in parallel
4     if R[i].A = R[i − 1].A then
5         R[i].A ← R[i].B
6 return The tuples of R
```

Lemma 4.1. *Let $G'(V', E')$ be the reduced graph of $G(V, E)$. Each connected component of G' forms a tree with only one leaf node, except that there may be an extra self-loop at the root node of the tree. For any node $v \in V'$, $v \notin S$ if and only if v is the leaf of a tree with root $r \leq n - s$.*

Proof. In the reduced graph G', it can be easily verified that the in-degree of any node is at most 1, and the out-degree of any node is also at most 1 if ignoring all self-loops. Therefore, each connected component of G' forms a tree with only one root node, except that there may be an extra self-loop at the root node of the tree. Let $v_1, v_2, \ldots, v_k$ be the nodes of such a tree from root v_1 to the only leaf v_k, so $v_1 < v_2 < \cdots < v_k$. We are going to prove that (1) if $(v_1, v_1) \notin E'$, then $v_i \in S$ for any $i < k$, while $v_k \notin S$; (2) if $(v_1, v_1) \in E'$, then $v_i \in S$ for all i; (3) $(v_1, v_1) \notin E'$ if and only if $v_1 \leq n - s$. By combining the three results we conclude the proof.

First we prove (1), in which case $(v_1, v_1) \notin E'$. Since $(v_1, v_2) \in E$, we consider in the algorithm with the loop when $a = v_1$ and $b = v_2$. In this time $v_1 \notin S$ so the algorithm takes v_1 to S and ignore v_2, because otherwise there exists some $v < v_1$ that $(v, v_1) \in E$, then it implies $(v_1, v_1) \in E'$ due to the construction of G'. Therefore, by the algorithm v_1 will be taken to the sample. Then we move to the loop when $a = v_2$ and $b = v_3$, which runs after $a = v_1$. Since v_2 has only appeared once and was ignored, $v_2 \notin S$, so the algorithm takes v_2 to S and ignore v_3, and so on, until we move to the loop with $a = v_{k-1}$ and $b = v_k$, and the algorithm takes v_{k-1} to S and ignore v_k. Since the out-degree of v_k is 0, there is no more chance that v_k is sampled. Therefore, we conclude (1).

Next we prove (2). Since $(v_1, v_1) \in E'$, either $(v_1, v_1) \in E$ or there exists $v < v' < v_1$ such that $(v, v') \in E$ and $(v, v_1) \in E$. In the former case, v_1 is taken to S when $a = b = v_1$. In the latter case, the algorithm meets $a = v, b = v'$ before $a = v, b = v_1$. In the time $a = v$ and $b = v'$, either $a \in S$ already, or a is taken to S at this step, so in any case, $a \in S$ after this step. Therefore, in the time $a = v$ and $b = v_1$, v_1 is taken to S because v is already in S. We then move to $a = v_1$ and $b = v_2$, and

find that v_1 is already in S so v_2 is taken to S, and so on, until we move to the loop with $a = v_{k-1}$ and $b = v_k$, and the algorithm takes v_k to S because v_{k-1} is already in S. Therefore, we conclude (2).

Recall that the in-degree of any node in G' is at most 1. Also note that during the construction of the reduced graph, the in-degree of any node does not change. Therefore, we prove (3) by noting the fact that $(v_1, v_1) \in E'$ if and only if the in-degree of v_1 is 1 in G', hence is 1 in G, if and only if $v_1 = b_i$ for some i, if and only if $v_1 > n - s$.

By combining the results of (1)(2)(3) we conclude the lemma. □

From the lemma we can easily verify that S is the set of nodes V' in the reduced graph G' except those leaf nodes with root at most $n - s$. To correctly connect the leaf node to its root, we will need the pointer jumping technique introduced below.

4.2 Pointer Jumping

Next we introduce the *pointer jumping* technique. It is a standard technique in parallel computing to find the roots of all nodes, and can be used in the MPC model. The basic operation of pointer jumping is to replace each tail of a directed edge to its tail's (only) tail, if it exists. That is, for two consecutive direct edge (u, v), (v, w), pointer jumping changes edge (v, w) to edge (u, w). See Figure 1(c) for an example. In one round, every pair of consecutive edges is updated simultaneously. It is easy to verify that after at most $h = \lceil \log d \rceil$ jumps, there is an edge between any node's original root and itself, where d is *diameter*, i.e., the length of the longest tree in the graph. Each jump can be directly realized by a primary key join circuit, as shown in Algorithm 5.

Algorithm 5: Pointer jumping circuit

Input: Edges $\{(a_i, b_i)\}_{i=1}^{s}$ with all b_i distinct; number of jumps h
Output: Updated edges after jumping

Initialize relation $R(A, B)$ with tuples $\{(a_i, b_i)\}_{i=1}^{s}$;
for $i \leftarrow 1$ **to** h **do**
  $S(B, C) \leftarrow R(A, B)$ with columns (A, B) renamed to (B, C);
  $T(A, B, C) \leftarrow R \bowtie S$, where B is the primary key of R;
  for $j \leftarrow 1$ **to** s **do in parallel**
    if $T[j].A = \perp$ **then**
      $T[j].A \leftarrow T[j].B$; `// Leave the edge unchanged if it cannot jump`
  $R(A, C) \leftarrow T(A, B, C)$ with column B removed;
  Rename column C of $R(A, C)$ to B;
return *The tuples of R*

4.3 The Batch WoR Sampling Circuit

Now we are ready to introduce the circuit for generating the indices of a WoR sample in Algorithm 6. First we follow the Floyd's algorithm to get $\{(a_i, b_i)\}_{i=1}^{n}$ which is also the set of edges in the original graph G. The we reduce the graph to get G' by Algorithm 4. We then link the nodes of G' to their roots by Algorithm 5, and finally carefully choose the set of sample S by Lemma 4.1. The size and depth of the circuit are dominated by the size and depth of the pointer jumping circuit, which are $O(sh \log^2 s)$ and $O(h \log^2 s)$ respectively. By taking $h = \log s$, the size and depth are $O(s \log^3 s)$ and $O(\log^3 s)$ respectively. We give an example in Figure 1, in which $n = 20$ and $s = 8$.

Algorithm 6: Generate the indices of a WoR sample

Input: Domain size n; number of jumps h
Output: The indices S of a WoR sample

Initialize $R(A, B)$ with size s ;
for $i \leftarrow 1$ **to** s **do in parallel**
  $b_i \leftarrow n - s + i$;
  $a_i \leftarrow \text{URNG}(b_i)$;
$E \leftarrow \{(a_i, b_i)\}_{i=1}^{s}$;
$E' \leftarrow$ the output of Algorithm 4 with input E;
$E' \leftarrow$ the output of Algorithm 5 with input E' and h;
$R(A, B) \leftarrow E'$;
Sort R by (A, B);
Initialize an array S with size s;
for $i \leftarrow 1$ **to** s **do in parallel**
  $(a, b) \leftarrow (R[i].A, R[i].B)$;
  if $a \le n - s$ **and** $(i = s$ **or** $a \ne R[i + 1].A)$ **then**
    $S[i] \leftarrow a$;
  else
    $S[i] \leftarrow b$;
return S

4.4 Reduce the Number of Jumps

Although d, the diameter of G', can be as large as s in the worst case, this happens with negligible probability. If one does not pursue an algorithm that always succeeds and returns a perfect uniform sample, both the size and the depth of the circuit can be reduced by setting a smaller h.

Lemma 4.2. *For any $m \ge 2e \ln n$, the diameter d of the reduced graph G' is larger than m with probability at most $n \cdot 2^{-m}$.*

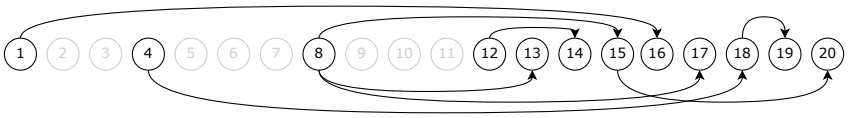
Proof. Let G_j be the subgraph of G, obtained by running the Floyd's algorithm for only the first j steps, i.e., $G_j = (V, E_j)$ where $E_j = \{(a_i, b_i)\}_{i=1}^{j}$, $b_i = n - s + i$ and a_i is taken from $\text{URNG}(b_i)$. Let $G'_j = (V'_j, E'_j)$ be the reduced graph of G_j. While inserting (a_{j+1}, b_{j+1}) into E_j we easily obtain G_{j+1} from G_j, we can also verify that by inserting a node b_{j+1} to V'_j and an edge (v, b_{j+1}) to E'_j, we obtain G'_{j+1} from G_j, where $v = b_{j+1}$ if the out-degree of a_{i+1} in G'_j is at least 1, and $v = a_{j+1}$ otherwise.

Let $l_j(i)$ be leaf node of the tree starting from node i in G'_j, ignoring all self-loops, while setting $l_j(i) = 0$ if such tree does not exist, i.e., i is not the root of any tree. We also define $d_j(i)$ as the height of this tree, and $d_j(i) = 0$ if this tree does not exist. Moving to the $j + 1$-th step, the height of the tree is increased by 1 if and only if $l_j(i) < b_{j+1}$ and $a_{j+1} = l_j(i)$. Given $l_j(i) < b_{j+1}$, the event $a_{j+1} = l_j(i)$ happens with probability $1/(n - s + j + 1)$. Therefore,

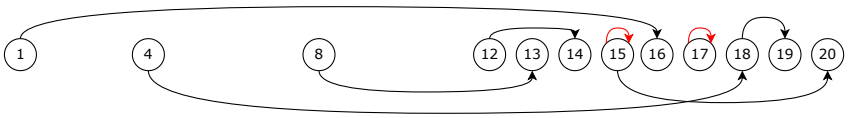
$$\Pr[d_{j+1}(i) = d_j(i) + 1 \mid G'_j] \le 1/(n - s + j).$$

Let Y_i be a Bernoulli variable with parameter $1/(n-s+i)$, and $\mathbb{E}[\sum_{i=1}^{s} Y_i] = \sum_{i=1}^{s} 1/(n-s+i) \le \ln n$. Let $\delta = m/\ln n \ge 2e$. By Chernoff bound,

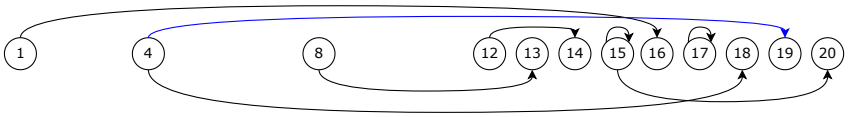
$$\Pr\left[\sum_{i=1}^{s} Y_i \ge m\right] \le \left(\frac{e^{\delta-1}}{\delta^{\delta}}\right)^{\ln n} < 2^{-m}.$$



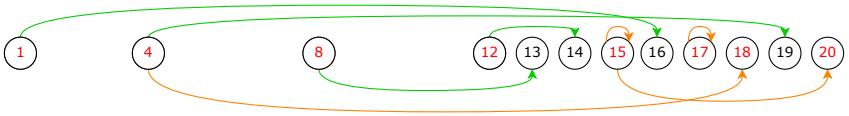
(a) The original graph G with edges $E = \{(8, 13), (12, 14), (8, 15), (1, 16), (8, 17), (4, 18), (18, 19), (15, 20)\}$.



(b) The reduced graph G' of G, where the edges $(8, 15)$ and $(8, 17)$ are replaced by the self-loops (marked in red) $(15, 15)$ and $(17, 17)$, respectively.



(c) After applying pointer jumping to G', the edge $(18, 19)$ is replaced by $(4, 19)$ and colored in blue.



(d) The edges that we take the heads to S are $(4, 18), (15, 15), (15, 20), (17, 17)$ (orange lines); we take the tails of other edges to S (green lines). Finally $S = \{1, 4, 8, 12, 15, 17, 18, 20\}$.

Fig. 1. Example of Algorithm 6

This implies $\Pr[d_s(i) \geq m] \leq 2^{-m}$. Note that the diameter of G' is the maximum diameter of its connected components, which is the maximum height of the trees. Therefore, by union bound,

$$\Pr[d \geq m] = \Pr\left[\bigvee_{i=1}^{n} d_s(i) \geq m\right] \leq \sum_{i=1}^{n} \Pr[d_s(i) \geq m] \leq n \cdot 2^{-m}. \qquad \square$$

Corollary 4.3. *Taking $h = \lceil \log(\sigma + 2e \cdot \ln n) \rceil$, Algorithm 6 fails with probability at most $2^{-\sigma}$, where σ is the statistical security parameter and $\sigma = \Omega(\log n)$. If it succeeds, then it returns a sample of $[n]$ with statistical distance to a perfect WoR sample with samplei size s at most $2^{-\sigma}$. Recall that we assume $\sigma = \Omega(\log n)$, so $h = O(\log \sigma)$. Therefore, the circuit has size $O(s \log^2 s \log \sigma)$ and depth $O(\log^2 s \log \sigma)$.*

Finally we conclude the main theorem of this section.

Theorem 4.4. *There exists a circuit that generates a batch of WoR samples with size $O(n \log^2 n \log \sigma)$ and depth $O(\log^2 n \log \sigma)$. It fails with probability at most $2^{-\sigma}$.*

5 STRATIFIED SAMPLING

In stratified sampling, the elements are divided into g strata, where the j-th stratum has size d_j. Stratified sampling is very useful for group-by queries, where each group is a stratum. Suppose the data is given in a relation R that is stratified by attribute *gid* (i.e., *gid* is the group-by attribute), whose values are taken from $[g]$. Let k_j be the number of elements sampled from the j-th stratum. If g, d_j, k_j are all public, stratified sampling simply reduces to g instances of WoR sampling, as

done in SAQE [9]. Here, we aim at full privacy protection where g, d_j, k_j are all kept private (e.g., g might be the number of customers and the d_j's are the numbers of orders placed by the customers), namely, the only information released is the input size n and the sample size s (from all strata). It turns out we can achieve full-privacy protection for stratified sampling using a circuit whose size is the same as that of WoR sampling, which is the special case when there is just one stratum.

5.1 Sizing Policy

How to set the sample size k_j for each stratum depends on the stratified sample sizing policy, and a technical challenge for full-privacy protection is to compute these k_j's using a circuit under a given policy. We consider the following two common policies.

Individualized sample sizes. Under this policy, one sets $k_j = F(j, d_j)$ for a certain function F such that $F(j, d_j) \le d_j$ and $s = \sum_{j=1}^{g} F(j, d_j)$. For example, $F(j, d_j) = d_j \cdot s/n$ yields a uniform policy where the sample size is proportional to the stratum size; other forms of $F(j, d_j)$ may yield a non-uniform policy that emphasizes certain strata. We assume $F(j, d_j) \ge 1$ for all j, which implies $g \le s$.

Given the relation R, we compute the k_j's by Algorithm 7. Note that g is private and can be as large as s, so we output s values $d_1, \ldots, d_s$, where d_i is the i-th stratum size if $i \le g$, and $d_i = \perp$ otherwise. Then the sample sizes $(k_1, \ldots, k_s)$ can be computed by F. Note that if $d_i = \perp$, then $k_i = 0$, i.e., does not take any element from a dummy stratum.

Algorithm 7: Compute strata sizes and sample sizes

Input: $R(eid, gid)$
Output: Strata sizes $(d_1, \ldots, d_s)$; sample sizes $(k_1, \ldots, k_s)$

```
Sort R by gid;
(d_1, ..., d_n) ← prefix-+ of (1, 1, ..., 1) segmented by R.gid;
for i ← 1 to n − 1 do in parallel
    if R[i].eid = R[i + 1].eid then
        d_i ← ⊥;
Compact (d_1, ..., d_n) to size s;
for i ← 1 to s do in parallel
    if d_i = ⊥ then
        k_i ← 0;
    else
        k_i ← F(i, d_i);
return (d_1, ..., d_s), (k_1, ..., k_s)
```

Threshold policy. Another common sizing policy (e.g., adopted in BlinkDB [1]) first finds a threshold k and then sets $k_j = \min(k, d_j)$, where k is the maximum integer such that $\sum_{j=1}^{g} k_j \le s$. In other words, those strata with size $\le k$ will be fully taken to the sample, while for any other stratum, a WoR sample of size k will be taken. This policy has the benefit that all groups are well represented in the sample. Different from the individualized sizing policy, in threshold policy, k_i does not only depend on j and d_j, but all the d_j's.

Next we discuss how to compute k, hence all the k_j's for the threshold policy. Let $\{d_j\}_{j=1}^{s}$ be the strata sizes computed by Algorithm 7, where $s - g$ of them are dummy. We first sort $\{d_i\}_{i=1}^{g}$

in ascending order (putting dummy elements at the end). Abusing notation, we still let $\{d_i\}_{i=1}^{s}$ be sequence after sorting, so that $d_1 \le d_2 \le \cdots \le d_g$ and $d_{g+1} = \cdots = d_s = \perp$. Let t_j be the total sample size of all strata when $k = d_j$, then

$$t_j = \begin{cases} d_1 \cdot g, & j = 1; \\ t_{j-1} + (d_j - d_{j-1}) \cdot (g - j + 1), & 2 \le j \le g; \\ \perp, & \text{otherwise.} \end{cases}$$

Let l be the maximum value such that $s \ge t_l$, which can be computed by a prefix-sum circuit and then taking the last value. Then

$$k = \lfloor (s - t_l)/(g - l) \rfloor + d_l.$$

Once k is computed, the sample sizes of the strata is are defined by setting $k_j = \min(d_j, k)$. Our circuit for computing the threshold k is described in Algorithm 8, in which the binary operator $\oplus$ is defined as

$$(l_1, d_1, t_1) \oplus (l_2, d_2, t_2) = \begin{cases} (l_2, d_2, t_2), & \text{if } l_2 \ge l_1; \\ (l_1, d_1, t_1), & \text{otherwise.} \end{cases}$$

The circuit has size $O(s \log^2 s)$ and depth $O(\log^2 s)$.

Algorithm 8: Compute threshold

Input: Strata sizes $(d_1, \ldots, d_s)$
Output: The threshold k

$(d_1, \ldots, d_s) \leftarrow$ `Sort` $(d_1, \ldots, d_s)$ while putting $\perp$ at the end;
$t_1 \leftarrow d_1 \cdot g$;
for $j \leftarrow 2$ **to** s **do in parallel**
  $t_j \leftarrow (d_j - d_{j-1}) \cdot (g - j + 1)$;
$(t_1, \ldots, t_s) \leftarrow$ `prefix-+` of $(t_1, \ldots, t_s)$;
Initialize a relation $R(L, D, T)$ with size s;
for $j \leftarrow 1$ **to** s **do in parallel**
  if $j \le g$ **and** $t_j \le s$ **then**
    $R[j].L \leftarrow j$;
  else
    $R[j].L \leftarrow 0$;
  $R[j].D \leftarrow d_j$;
  $R[j].T \leftarrow t_j$;
$R \leftarrow$ `prefix-`$\oplus$ of R;
$(l, d, t) \leftarrow R[s]$;
$k \leftarrow \lfloor (s - t)/(g - l) \rfloor + d$;
return k

5.2 The Batch Stratified Sampling Circuit

Our circuit for generating the indices of a stratified sample is described in Algorithm 9. Note that the indices correspond to $\mathcal{X}$ after sorting by their strata ids. In this algorithm, $gid, R.D, R.K$ are the ids, sizes, sample sizes of the strata, respectively. $R.O$ stands for the index of an element in

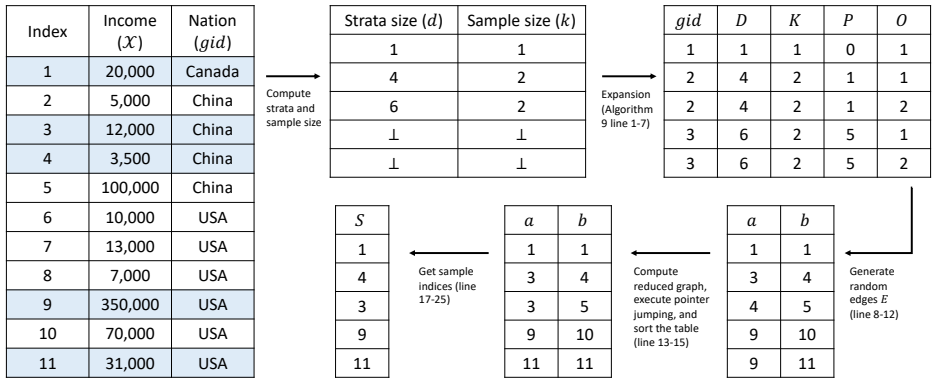


Fig. 2. Example of Algorithm 9

its stratum, its value plus the corresponding $R.P$ yields the index to the original element in $\mathcal{X}$. When generating the indices (line 8–24), we use the same solution that we generate indices for a WoR sample, except that the ranges of the uniform number generators are also determined by the strata sizes. See Figure 2 for an example of the circuit, in which there are three groups of incomes, specified by the nation. In the example, we generate the indices of a stratified sample from three strata with sizes 1,6,6 and sample sizes 1,2,2 respectively. The final output is $S = \{1, 3, 4, 9, 11\}$. Finally we conclude the main theorem of this section.

Theorem 5.1. *There exists a circuit of size $O(n \log^2 n \log \sigma)$ and depth $O(\log^2 n \log \sigma)$ that generates a batch of stratified samples under the individualized and the threshold policy. It fails with probability at most $2^{-\sigma}$.*

6 MASQUE: A SAMPLING-BASED MPC-AQP SYSTEM

6.1 System Overview

We build MASQUE, our sampling-based two-stage MPC-AQP system in the two-server model. The two-server model splits the input parties (i.e., data owners) and computing parties by introducing two semi-honest, non-colluding servers for computation. Data owners do not trust any single party, and distribute trust across the two servers by secret-sharing their private data. Data are contributed by any number of data owners, which may partition the data either horizontally (each data owner contributes a subset of tuples of the same table), vertically (each data owner contributes a different table), or in a mixed fashion. Note that the two-server model is more general than the two-party model in [53] (where the two servers are also the two data owners) and more secure and easier to deploy than the three-server honest majority model [27]. And it reduces the pairwise secure communication costs between data owners if all data owners are involved in the computation.

During the offline stage, MASQUE performs the following tasks:

(1) The data owners compute the secret shares of their data (e.g., Boolean shares), and send the shares to the two servers, respectively.
(2) Using any secure join protocol that works under the two-server model [7, 54], the two servers denormalize the data and obtain a flat table in secret-shared form.
(3) Evaluate the batch sampling circuits described above on the flat table to prepare a batch of samples.

During the online stage, the following happens with each query:

Algorithm 9: Generate the indices of a stratified sample

Input: Strata sizes $(d_1, \ldots, d_s)$; sample sizes $(k_1, \ldots, k_s)$
Output: The indices S of a stratified sample

$(p_1, \ldots, p_s) \leftarrow$ `prefix-+` of $(0, d_1, \ldots, d_{s-1})$;
$gid \leftarrow$ `Expand` $(1, 2, \ldots, s)$ by $(k_1, \ldots, k_s)$;
Initialize a relation $R(D, K, P, O)$;
$R.D \leftarrow$ `Expand` $(d_1, \ldots, d_s)$ by $(k_1, \ldots, k_s)$;
$R.K \leftarrow$ `Expand` $(k_1, \ldots, k_s)$ by $(k_1, \ldots, k_s)$;
$R.P \leftarrow$ `Expand` $(p_1, \ldots, p_s)$ by $(k_1, \ldots, k_s)$;
$R.O \leftarrow$ `prefix-+` of $(1, \ldots, 1)$ segmented by gid;
for $i \leftarrow 1$ **to** s **do in parallel**
    $(d, k, p, o) \leftarrow R[i]$;
    $b_i \leftarrow d - k + o + p$;
    $a_i \leftarrow \mathrm{URNG}(d - k + o) + p$;
$E \leftarrow$ the output of Algorithm 4 with input $\{(a_i, b_i)\}_{i=1}^{s}$;
$E' \leftarrow$ the output of Algorithm 5 with input E' and $h = \lceil \log(\sigma + 2e \cdot \ln n) \rceil$;
$\{(a_i, b_i)\}_{i=1}^{s} \leftarrow$ `Sort` E' by $\{a_i\}$ then $\{b_i\}$;
Initialize an array S with size s;
for $i \leftarrow 1$ **to** s **do in parallel**
    $(d, k, p, o) \leftarrow R[i]$;
    if $a_i \le p + d - k$ **and** $(i = s$ **or** $a_i \ne a_{i+1})$ **then**
        $S[i] \leftarrow a_i$;
    else
        $S[i] \leftarrow b_i$;
return S

(1) A client (who might be one of the data owners) submits the query to one of the servers with an optionally specified sample size.
(2) The server forwards the query to all data owners for validation, and the data owners decide if differential privacy (DP) should be applied to sanitize the query result. For example, in Example 1.1, if the query asks for a particular patient's medical record, the hospitals can reject answering the query. If the query asks for some aggregates, then the hospitals can choose to answer the query with an appropriate differential privacy parameter ε and global sensitivity GS.
(3) The two servers construct a query evaluation circuit, and evaluate it on a sample, obtaining the query result in secret-shared form. When all samples are depleted, a new batch is generated.
(4) If DP is required by the data owners, the two servers build another circuit to inject noise to the query result.
(5) The two servers send the shares of the query result to the client for reconstruction.

6.2 Security Guarantee

MASQUE defends against a semi-honest, computationally bounded adversary, who might corrupt any number of data owners, the client, and at most one server. We provide full-fledged protection before, during, and after query processing: as long as the two servers do not collude, our system

guarantees that all parties learn nothing, except that the client will learn the output of the query, which could be further protected by DP.

The ideal functionality for MASQUE are presented in Functionality 10 (offline stage) and Functionality 11 (online stage), where we use $[\![\cdot]\!]$ to denote a value that is presented in secret-shared form. Since the input and output of the functionality are in secret-shared form, the servers actually learn nothing (including input data, sample data, and query results) during the protocol.

Algorithm 10: Offline Ideal Functionality $\mathcal{F}_{\text{offline}}$

Input: $[\![\mathcal{X}]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$, sample size s
Output: A batch of samples $[\![\mathcal{S}]\!] = ([\![S_1]\!], \ldots, [\![S_{n/s}]\!])$, each with sample size s

Recover $\mathcal{X}$ from $[\![\mathcal{X}]\!]$;
Take a batch of samples $\mathcal{S}$ from $\mathcal{X}$;
Compute the secret share $[\![\mathcal{S}]\!]$ of $\mathcal{S}$;
return $[\![\mathcal{S}]\!]$

Algorithm 11: Online Ideal Functionality $\mathcal{F}_{\text{online}}$

Input: Aggregate query Q; a batch of samples $[\![\mathcal{S}]\!]$; DP parameter ε, GS
Output: Query result $[\![q]\!]$

$[\![S]\!] \leftarrow$ a sample from $[\![\mathcal{S}]\!]$;
Remove $[\![S]\!]$ from $[\![\mathcal{S}]\!]$;
Recover S from $[\![S]\!]$;
$q \leftarrow Q(S) + Lap(\frac{GS \cdot s}{\varepsilon n})$;
Compute the secret share $[\![q]\!]$ of q;
return $[\![q]\!]$

6.3 MPC Protocols Optimization

MASQUE mostly uses Yao's garbled circuit protocol to evaluate a given circuit, but with the following optimizations:

(1) As mentioned previously, there is a more efficient sorting protocol under the two-server model [26]. So we replace each sorting component of the circuit with this protocol.
(2) The garbled circuit protocol operates on the bit level. For Boolean operations, such as AND and XOR, the GMW protocol is more efficient. Thus, when facing such operations, we invoke GMW through ABY [18].
(3) To evaluate bitwise AND gates more efficiently, we also generate Beaver multiplication triples [10] in the offline stage, which are then consumed during an online query.

6.4 Online Query Evaluation

The online query evaluation process of MASQUE largely follows SAQE [9], except that the samples are generated in batches. Specifically, we take the following steps for each online query:

(1) The query is first parsed. If it is a group-by query, a stratified sample is used; otherwise a WoR sample is used.
(2) A query plan is formed that consists of a projection, which prunes unnecessary columns, a selection operator with the predicates in the `WHERE` clause of the query, and an aggregation operator (possibly with a group-by).

(3) A circuit is generated following the query plan, which is then evaluated using a combination of garbled circuits and GMW.
(4) If differential privacy is required, the two servers construct a circuit to generate a noise and add it to the query result. There is a lot of research discussing how to properly compute a noise under DP [19–21, 28, 34, 51], which we can adopt in our system. In particular, we adopt the following scheme. Let GS be the global sensitivity of the query and we use a WoR sample with size s to answer the query. Then we add a noise drawn from a Laplacian distribution with scale parameter $\frac{GS \cdot s}{\varepsilon n}$ to obtain an ε-DP query result.
(5) If the aggregation function is `count` or `sum`, we need to scale up the result by a factor of n/s, i.e., if we sample 10% data for `count`, the query result on the sample will be roughly $1/10$ of that on the original data. So we need scale up the result by 10 for an unbiased estimation. Note that this can be done by the client in plaintext since n, s are public parameters. However, for group-by queries, the scaling factor is d_i/k_i, which is private. Then we use a garbled division circuit [18, 55] to re-scale the query result.

7 EXPERIMENTS

7.1 Experimental Setup

We compare MASQUE[5] with SAQE [9], SMCQL [7], and SecYan [53]. SAQE is the only MPC-AQP system in the past, while the latter are exact MPC query engines. All experiments were performed in a LAN setting, with a small network delay (around 0.1ms) and high bandwidth around 1Gb/s. The running times are measured on two servers, each equipped with 48 Intel Xeon Silver 4116 CPUs (but only one thread is used) and a large enough memory that can contain all the data. This models the common situation where the two non-colluding servers are hosted by two major cloud providers with a dedicated link. The security parameters are $\sigma = 40$ and $\kappa = 128$ (that is, 128-bit encryption key). The bit length of all attributes is 32. All the results are the average over 10 repetitions.

7.2 Offline: Sample Generation

We compare our sample generation circuits against that used in SAQE [9], which generates a Poisson sampling of size s with compaction. SAQE generates one sample with cost $O(n \log n)$, while our amortized per-sample cost is $O(s \log n)$ (for shuffle sampling and WR sampling) or $O(s \log n \log \sigma)$ (for WoR sampling and stratified sampling). Note that SAQE also provides stratified sampling, but it simply runs the sampling algorithm on each stratum, thereby revealing the number of strata and strata sizes, while our stratified sampling algorithm does not.

Table 2 shows the amortized sampling costs of different algorithms for sample size $s = 50$ and various database size n. Note that the costs of MPC protocols by definition are independent on the data, so the actual contents of the inputs do not affect the results. Shuffle sampling always performs the best in both time and communication cost, due to its simplicity. WR sampling performs 5 times better than WoR sampling in terms of time cost, because of more rounds of `sort` from pointer jumping circuit. Stratified sampling has a similar cost as WoR sampling. Figure 3 shows the amortized cost per sample for a fixed data size while varying the sample rate s/n. We see that SAQE has a fixed cost no matter how small the sample size is. It performs well when sample rate is large, i.e., greater than 5%. However, recall that the sampling error depends on the absolute sample size s (see Table 1), so the sampling rate for achieving a desired error reduces for large data sets.

Methods	$n = 10^3$		$n = 10^4$		$n = 10^5$	
	Time (s)	Com. (MB)	Time (s)	Com. (MB)	Time (s)	Com. (MB)
Shuffle Sampling	0.0675	2.67	0.0765	3.68	0.1	5.75
Sampling With Replacement	0.795	44.8	1.02	57.5	1.42	72.9
Sampling Without Replacement	3.88	165	5.50	212	9.65	273
Poisson Sampling [9]	3.49	25.3	42.1	254	665	2580
Stratified Sampling	3.93	165	5.60	215	9.75	270

Table 2. Amortized time and communication costs per sample of different methods on simulated dataset with different sizes

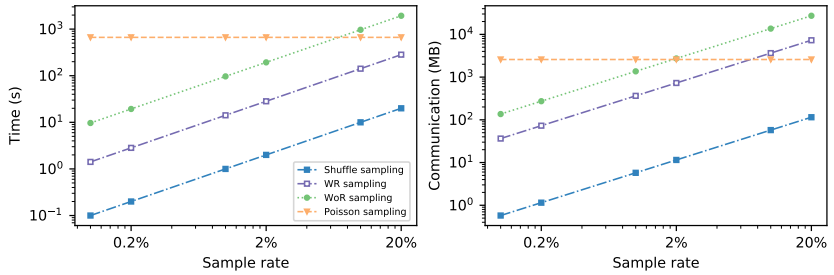


Fig. 3. Time and communication costs per sample with different sample rates

7.3 Online: Query Evaluation

Next, we compare the online query processing costs. We evaluated a 10MB TPC-H dataset containing 60,000 tuples in total, such that the baseline methods can still finish in a reasonable amount of time, and reported four representative queries:

- Q1. It computes 6 aggregates (**count** and **sum**) over the `lineitem` table grouped by `returnflag` and `linestatus`. The query also contains a few **avg** aggregates, which can be easily obtained from **count** and **sum**.
- Q1U. This is simplified version of Query 1 where we remove the group-by operator.
- Q8. This is a complicated analytical query involving selections, joins, and group-by aggregations. All the joins have been done in the offline stage, while the online stage computes the query with the specified selection conditions (i.e., the nation and region).
- Q9. This query calculates the **sum** of profits grouped by the nations.

We compare the online query processing costs of MASQUE against SMCQL [7], SAQE [9], and SecYan[53]. For SMCQL and SAQE, all joins are also performed in an offline stage to denormalize the data. On the other hand, SecYan performs the joins online. We also tested the processing costs over plain text (using MySQL) for benchmarking.

(1) SMCQL [7]: SMCQL builds and evaluates a garbled circuit to compute the query result exactly.
(2) SAQE [9]: It first uses a `compaction` circuit to extract samples, then evaluates a garbled circuit on the sample. Note that for group-by queries (Q1, Q8 and Q9), SAQE provides a weaker security protection, because its stratified sampling algorithm reveals the sizes of the strata (after adding DP noise). On the other hand, the stratified sampling algorithm in MASQUE does not reveal anything other than the query result (after adding DP noise).

[5]https://github.com/hkustDB/MASQUE

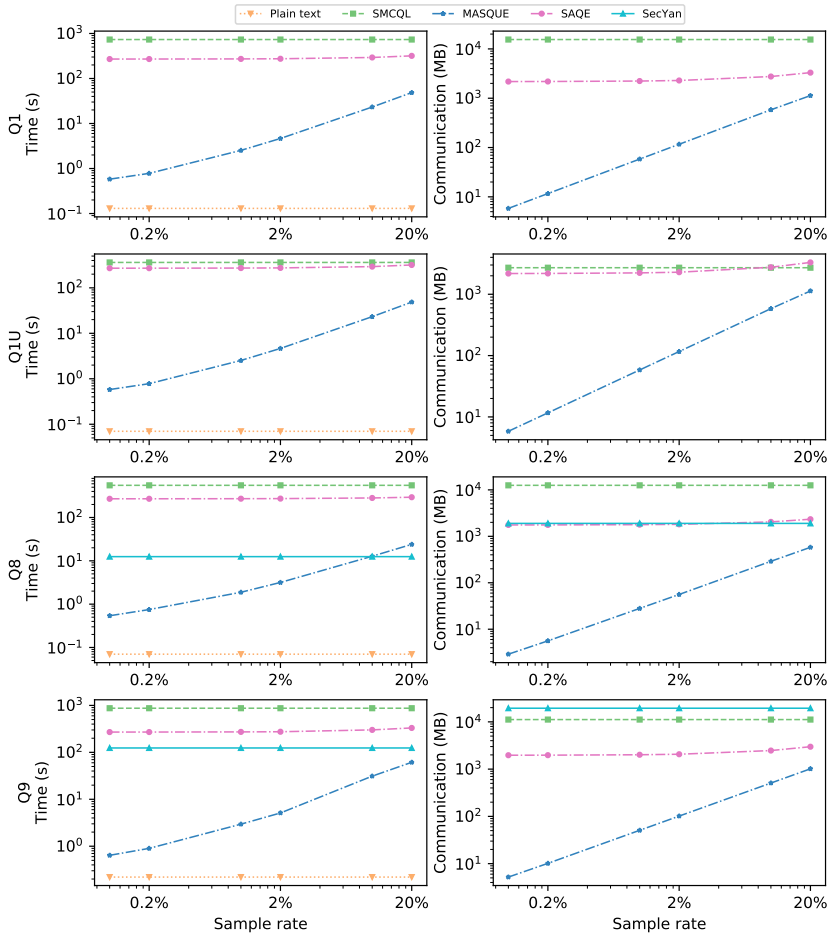


Fig. 4. Time and communication costs with different sample sizes on TPC-H

(3) SecYan [53]: SecYan is a secure two-party protocol especially designed for computing free-connex join-aggregate queries exactly. Q1 and Q1U do not have joins, so SecYan degenerates into the same garbled circuit as used in SMCQL. Q8 is a free-connex query. Q9 is not per se, and SecYan decomposes it into 25 such queries, one for each group (nation).

We experimented with sampling rates from 0.1% to 20% for MASQUE and SAQE. Figure 4 shows the time and communication costs of these systems. Note that both axes are drawn in log scale. As expected, MASQUE's costs are proportional to the sample size, where the other systems are not affected by the sampling rate: The exact query processing engines are not affected by the sampling rate by definition. The sampling cost of SAQE is $O(n \log n)$ and independent of the sample size, which is followed by an $O(s)$ query processing cost. Since the sampling cost dominates, the total cost is not significantly affected.

From the results, we see that the two MPC-AQP systems mostly outperform the exact MPC engines, except that SecYan is better than SAQE on Q8, which is a free-connex query that SecYan specializes on. However, the advantage of SAQE over SMCQL or SecYan is not obvious. Recall

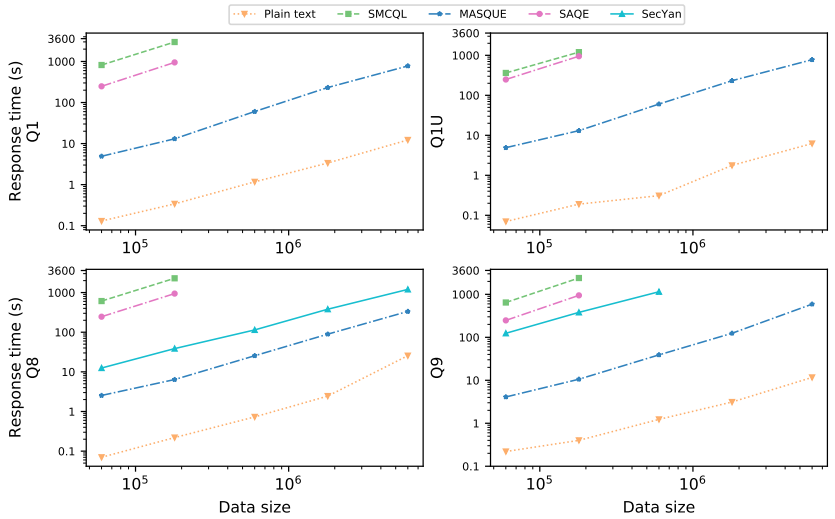


Fig. 5. Response time with different data sizes on TPC-H

that the former has an asymptotic cost of $O(n \log n)$ while the latter computes the exact query result with $O(n)$ cost but with a larger hidden constant. On the other hand, MASQUE significantly outperforms SAQE and the exact query processing engines, especially for smaller sampling rates, truly exploiting the accuracy-cost trade-off for sampling-based AQP.

We also evaluated the response time (i.e., the online query time) on different scales of data. We generated different of TPC-H datasets, with sizes varying from 10MB to 1GB, containing 6×10^4 to 6×10^6 tuples. Sample rate is set to 2% for two MPC-AQP systems (MASQUE and SAQE). We only report the response time within an hour, as shown in Figure 5. We found that SMCQL and SAQE cannot scale to TPC-H 100MB dataset which contains 6×10^5 tuples. SecYan performed well in free-connex query (i.e., Q8), and cannot scale to larger dataset in non-free-connex query (i.e., Q9). MASQUE, on the other hand, can scale to 1GB data with 6×10^6 tuples. And our response time is always less than all other secure MPC engines. The results show the fact that our AQP-MPC system is suitable for large amounts of data.

8 CONCLUSIONS

In this paper, we have presented MASQUE, an MPC-AQP system based on random sampling. By separating the sampling into an offline and an online stage, we are able to significantly reduce the online query processing cost, which is important for interactive exploration of private data. One direction for future improvement is the offline cost. Currently, it takes an hour for MASQUE to generate a batch of WoR samples from a 10MB dataset. This is partly due to the strong security guarantee that MASQUE aims to achieve (i.e., revealing nothing beyond the DP-protected query result). With slightly weaker security guarantees, such as *differential obliviousness* [47], we believe it is possible to reduce this high offline cost, and this remains an interesting future direction.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16201819, 16205420, and 16205422, and by Alibaba Group through Alibaba Innovative Research Program. We thank the anonymous reviewers for valuable suggestions on improving the paper.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. 1983. An Θ(n log n) Sorting Network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. 1–9.

[3] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *3rd Symposium on Simplicity in Algorithms, SOSA 2020*, Martin Farach-Colton and Inge Li Gørtz (Eds.). 8–14.

[4] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA). 125–138.

[5] Borja Balle, Gilles Barthe, and Marco Gaboardi. 2018. Privacy Amplification by Subsampling: Tight Analyses via Couplings and Divergences. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 6280–6290.

[6] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*. 307–314.

[7] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: Secure Querying for Federated Databases. *Proceedings of the VLDB Endowment* 10, 6 (2017), 673–684.

[8] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: Efficient SQL Query Processing in Differentially Private Data Federations. *Proc. VLDB Endow.* 12, 3 (nov 2018), 307–320. https://doi.org/10.14778/3291264.3291274

[9] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-Preserving Approximate Query Processing for Data Federations. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2691–2705.

[10] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91*. 420–432.

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. 1–10.

[12] Jon Bentley and Bob Floyd. 1987. Programming Pearls: A Sample of Brilliance. *Commun. ACM* 30, 9 (1987), 754–757.

[13] Manuel Blum. 1983. Coin Flipping by Telephone a Protocol for Solving Impossible Problems. *SIGACT News* 15, 1 (1983), 23–27.

[14] Jeffrey Champion, abhi shelat, and Jonathan Ullman. 2019. Securely Sampling Biased Coins with Applications to Differential Privacy. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 603–614.

[15] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 511–519.

[16] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An Efficient Secure Three-Party Sorting Protocol with an Honest Majority. Cryptology ePrint Archive, Paper 2019/695. https://eprint.iacr.org/2019/695 https://eprint.iacr.org/2019/695.

[17] Seung Geol Choi, Dana Dachman-Soled, S. Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. 2022. Secure Sampling with Sublinear Communication. Cryptology ePrint Archive, Paper 2022/660. https://eprint.iacr.org/2022/660 https://eprint.iacr.org/2022/660.

[18] Cryptography and Privacy Engineering Group at TU Darmstadt. [n. d.]. A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. https://github.com/encryptogroup/ABY.

[19] Wei Dong, Juanru Fang, Ke Yi, Yuchao Tao, and Ashwin Machanavajjhala. 2022. R2T: Instance-optimal Truncation for Differentially PrivateQuery Evaluation with Foreign Keys. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*.

[20] Wei Dong and Ke Yi. 2021. Residual Sensitivity for Deferentially Private Multi-Way Joins. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*.

[21] Wei Dong and Ke Yi. 2022. A Nearly Instance-optimal Differentially Private Mechanism for Conjunctive Queries. In *Proceedings of the ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*.

[22] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[23] Ronald Aylmer Fisher and Frank Yates. 1953. *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company.

[24] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 218–229.

[25] Michael T. Goodrich. 2011. Data-Oblivious External-Memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 379–388. https://doi.org/10.1145/1989493.1989555

[26] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2013. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Information Security and Cryptology – ICISC 2012*. 202–216.

[27] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiangyang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1740–1753.

[28] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[29] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data*. 631–646.

[30] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1575–1590.

[31] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. 2015. Rapid Sampling for Visualizations with Ordering Guarantees. *Proc. VLDB Endow.* 8, 5 (2015), 521–532.

[32] Vladimir Kolesnikov and Ranjit Kumaresan. 2013. Improved OT Extension for Transferring Short Secrets. In *Advances in Cryptology – CRYPTO 2013*. 54–70.

[33] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 818–829. https://doi.org/10.1145/2976749.2978381

[34] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: a differentially private SQL query engine. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1371–1384.

[35] 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

[36] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838.

[37] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA). 615–629.

[38] Kaiyu Li, Yong Zhang, Guoliang Li, Wenbo Tao, and Ying Yan. 2019. Bounded Approximate Query Processing. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2019), 2262–2276. https://doi.org/10.1109/TKDE.2018.2877362

[39] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. 2014. Automating Efficient RAM-Model Secure Computation. In *2014 IEEE Symposium on Security and Privacy*. 623–638. https://doi.org/10.1109/SP.2014.46

[40] 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*. 1271–1287.

[41] Payman Mohassel and Saeed Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 557–574.

[42] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1461–1476.

[43] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient circuit-based psi with linear communication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 122–153.

[44] 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 *Proceedings of the 30th Conference on USENIX Security Symposium*.

[45] Manoj M Prabhakaran and Vinod M Prabhakaran. 2012. On secure multiparty sampling for more than two parties. In *2012 IEEE Information Theory Workshop*. 99–103.

[46] Vinod M Prabhakaran and Manoj M Prabhakaran. 2014. Assisted common information with an application to secure two-party sampling. *IEEE Transactions on Information Theory* 60, 6 (2014), 3413–3434.

[47] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. 2023. Differentially Oblivious Relational Database Operators. In *VLDB*.
[48] 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
[49] Sajin Sasy and Olga Ohrimenko. 2019. *Oblivious Sampling Algorithms for Private Data Analysis*.
[50] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *PODS*.
[51] Yuchao Tao, Xi He, Ashwin Machanavajjhala, and Sudeepa Roy. 2020. Computing Local Sensitivities of Counting Queries with Joins. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 479–494.
[52] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: Secure Multi-Party Computation on Big Data. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 18 pages. https://doi.org/10.1145/3302424.3303982
[53] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*.
[54] Yilei Wang and Ke Yi. 2022. Query Evaluation by Circuits. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*.
[55] Jonathan Katz Xiao Wang, Alex J. Malozemoff. [n. d.]. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
[56] Andrew C Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 160–164.
[57] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*. 162–167.

Received January 2023; revised April 2023; accepted May 2023