Longshot: Indexing Growing Databases using MPC and Differential Privacy

Yanping Zhang
Duke University
yanping.zhang@duke.edu

Johes Bater
Tufts Univeresity
johes.bater@tufts.edu

Kartik Nayak
Duke University
kartik@cs.duke.edu

Ashwin Machanavajjhala
Duke University
ashwin@cs.duke.edu

ABSTRACT

In this work, we propose Longshot, a novel design for secure outsourced database systems that supports ad-hoc queries through the use of secure multi-party computation and differential privacy. By combining these two techniques, we build and maintain data structures (i.e., synopses, indexes, and stores) that improve query execution efficiency while maintaining strong privacy and security guarantees. As new data records are uploaded by data owners, these data structures are continually updated by Longshot using novel algorithms that leverage bounded information leakage to minimize the use of expensive cryptographic protocols. Furthermore, Longshot organizes the data structures as a hierarchical tree based on when the update occurred, allowing for update strategies that provide logarithmic error over time. Through this approach, Longshot introduces a tunable three-way trade-off between privacy, accuracy, and efficiency. Our experimental results confirm that our optimizations are not only asymptotic improvements but also observable in practice. In particular, we see a 5x efficiency improvement to update our data structures even when the number of updates is less than 200. Moreover, the data structures significantly improve query runtimes over time, about $\sim 10^3$x faster compared to the baseline after 20 updates.

PVLDB Reference Format:
Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. Longshot: Indexing Growing Databases using MPC and Differential Privacy. PVLDB, 16(8): 2005 - 2018, 2023.
doi:10.14778/3594512.3594529

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ypzhang725/Longshot/tree/vldb.

1 INTRODUCTION

Secure database systems enable outsourcing data and computation to untrusted servers while preserving privacy. Several existing approaches achieve this goal. One class of techniques centers on utilizing cryptographically secure protocols, such as server-aided secure multi-party computation (MPC) [6], homomorphic encryption [21], or trusted hardware [31] on the server to execute analyst-provided queries. While these approaches ensure that the server does not gain plaintext access to private information, they utilize data-independent, worst-case query execution to prevent side channel leakage. This introduces an enormous performance cost, on the order of 1000x [6], which makes performance too costly to be practical. Another class of work [26, 35–37, 42, 42, 77, 80] provides practical performance by leaking access patterns and response volume (the result set size for a query). However, these approaches are susceptible to attacks that can reconstruct the private data distribution by observing the leakage [12, 18]. With this, we can see an inherent tension between privacy and efficiency.

Furthermore, this tension becomes more pronounced when considering practical issues such as growing data and multiple clients. When data grows, i.e., clients continually upload new records, secure database systems need to store and fetch these records without knowing their actual plaintext values. In the worst case, answering an ad-hoc query requires the server to carry out an expensive cryptographically secure scan over all records ever uploaded to fetch a specific record. If there is a single client as in Kellaris et al.[43], the client could provide a differentially private index to help the server avoid scanning all records every time. However, many scenarios have multiple clients uploading their private records. In this case, the index cannot be generated by a single client, thus necessitating the server to scan all uploaded records for each query.

Recent efforts [7, 8, 75, 76] strike a promising balance between privacy and performance by allowing bounded information leakage under differential privacy (DP) [30], but none of them satisfies all the requirements. These works improve the performance of query processing by hiding either access patterns or response volume only for specific and limited settings and do not generalize well to a practical setting that requires ad-hoc queries over growing data from multiple clients. Kellaris et al. uses ORAM [34, 69] to combat access pattern leakage and bound volume leakage using DP, but cannot support multiple clients continually uploading new private data. Mazloom et al. [54] hides access patterns of MPC using DP only in a static data setting. Shrinkwrap [7] reduces operator-level execution time of MPC through DP volume leakage, and IncShrink [76] generalizes Shrinkwrap to support updates continuously from multiple clients. However, IncShrink requires one specific query to be specified in advance. Using Shrinkwrap or IncShrink to support ad-hoc queries leads to a drastically increased and redundant computation cost for each query on all uploaded data. Furthermore,

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594529

repetitive querying of the same data can cause cumulative privacy loss, ultimately resulting in a complete loss of privacy [28].

In short, there is no prior privacy-preserving database management system that can support efficient and accurate ad-hoc queries on growing data from multiple clients. In this paper, instead of hiding one leakage source or improving the performance of processing one query using DP, we propose, Longshot, a system that constructs and continually updates differentially private data structures such that Longshot 1) can avoid redundant computation by using data structures computed during previous updates and 2) can process unlimited ad-hoc predicate queries in public using the data structures once they are released under differential privacy.

We summarize the contributions of Longshot as follows. The key technical ideas are described in more detail in Section 2.

- Longshot is a first of its kind, secure growing database system that maintains data structures to support ad-hoc predicate queries on data uploaded from multiple clients with provable security and privacy guarantees, as well as practical performance. Compared to executing queries without these structures, Longshot significantly improves both query efficiency and accuracy.
- Longshot introduces efficient oblivious algorithms to create differentially private statistics over growing data a.k.a. DP synopses on untrusted servers. The synopses are the only leakage source of access patterns, query volumes and update patterns when processing queries and updating data structures.
- Longshot introduces novel oblivious algorithms to store the data records efficiently without additional privacy leakage. In particular, our solution utilizes the differentially private statistics in the synopses to reduce the number of records on which computation needs to be performed using MPC from a linear factor to a logarithmic factor in the number of updates. Moreover, in the system, the data records layout map exactly with the synopses, due to which data records can be fetched using a publicly available index; thus leading to extremely efficient query processing.
- We evaluate Longshot on a real-world NYC taxi dataset [2]. The evaluation results show improved performance for updating the data structures and accuracy improvements over baseline approaches. Our optimization to update data stores achieves a 5x efficiency improvement even when the number of updates are less than 200 (and more subsequently since we perform computations over a poly-logarithmic amount of data records instead of a linear number of records). Moreover, the data structures significantly improve query runtimes over time, about $\sim 10^3$x faster compared to the baseline after 20 updates. Our evaluation shows how Longshot allows users to adjust the configuration of the system to ensure the desired guarantee and identify their desired trade-off between privacy, efficiency, and accuracy.

2 OVERVIEW

We design Longshot to achieve three main goals:

- **Efficient query execution through data structure maintenance over growing databases.** Longshot provides efficient query execution by maintaining privacy-preserving data structures that continuously, and securely, store and update uploaded records and statistics.
- **End-to-end privacy for multiple data owners against untrusted servers.** Longshot allows untrusted servers to execute queries on data uploaded from multiple data owners while enforcing strong privacy guarantees with bounded privacy loss.
- **Practical query performance and result accuracy.** Longshot maintains accurate results and efficient query execution of ad-hoc predicate queries with tunable trade-offs between privacy, accuracy, and performance.

In this section, we go over the key ideas of Longshot (Section 2.1), then discuss the required system components (Section 2.2), and end with a look at the overall system workflow (Section 2.3).

2.1 Key Ideas

KI-0. Secure databases with DP leakage. Privacy-preserving query processing schemes often allow information leakage of input data as a trade-off for efficiency. Recent works leverage differential privacy to bound the leakage of access patterns [11, 18, 39, 40, 42, 58, 64], update patterns [75] or intermediate/output volume [35, 37, 42, 45, 47] to provide a rigorous privacy and efficiency trade-off. However, these works hide only one type of leakage in a specific setting, which introduces strict limits on supported queries. Longshot aims to use a single leakage function to hide all these leakages and design privacy-preserving data structures to process ad-hoc queries on growing data from multiple data owners. The key insight is that Longshot measures data distribution using DP histograms and orders records by domain value such that the volume of records for each domain value is equal to its DP count. Longshot can use the DP histograms to either answer queries directly or as an index to fetch records from a DP store. The DP histograms account for all leakage types.

KI-1. DP synopsis updates over growing data. To support ad-hoc queries on growing data collected from multiple data owners, Longshot continually generates synopses over time. Each synopsis summarizes statistical information about data received over a specific time interval. Crucially, the synopses are constructed and released under a strict privacy loss bound, which introduces error in exchange for privacy. Once released, each synopsis can be used to answer arbitrary predicate aggregate queries, and improve efficiency when answering non-aggregate queries and updating DP stores (further explained in KI-2 and KI-4).

To ensure that query results have high accuracy, we design DP strategies that reduce errors in generated synopses using two techniques. First, we deploy a trigger-based update strategy to separate continuously uploaded data into disjoint blocks, so that Longshot can use existing differentially private statistical summary generation algorithms [46, 50, 55] on data blocks as a black box. The DP algorithms are designed to optimize outputs to answer a workload of linear queries on a single table with multiple attributes. In this way, Longshot can be further extended to support multiple attributes and tune the synopsis to provide better accuracy for a given workload. Second, naïvely applying DP algorithms on growing data accumulates error linearly w.r.t. the number of updates. Instead, we hierarchically update synopses to achieve logarithmic error in the number of updates. We further reduce error by using constrained inference [38, 63] to resolve inconsistencies in noisy measurements of different levels, without violate our desired privacy goals.

KI-2. DP store and DP index maintenance over growing data. In addition to answering aggregate queries, Longshot efficiently and securely returns records to answer non-aggregate queries through DP statistics. DP synopses provide these statistics, leaking a bounded amount of private information to improve the performance of fetching records. We develop novel indexing techniques to construct indexes and data stores that use DP synopses to lay out and store encrypted records. Specifically, DP synopses are used to construct and update an index that maintains a mapping between domain values and record positions in the data store. The data store contains the encrypted records described by the DP synopses. These records are ordered by their domain values such that the volume of records with each domain value is consistent with the DP synopses. The untrusted computing servers that hold the data store do not know the exact volume of records corresponding to each domain value as the records are encrypted. Using the DP synopses as the sole source of information about the volume and location of records with each domain value, Longshot bounds the privacy leakage when locating and fetching records on the server.

KI-3. Secure synopsis generation under MPC. In order to construct the data structures over data uploaded from multiple data owners on untrusted computing servers, Longshot utilizes MPC to securely process growing data without revealing records in the clear. While MPC is an effective tool, it introduces a performance overhead of up to three orders of magnitude [6], meaning that naïvely applying it to process all uploaded records at every time step can quickly become computationally infeasible. Instead, our approach ensures the efficiency of secure synopsis generation on growing data by minimizing the data and algorithm steps requiring MPC. The DP synopsis generation of Longshot using MPC improves performance by: 1) carrying out post-processing of computed noisy statistics, such as constrained inference [49] in the clear and 2) re-using previously computed statistics as inputs to future computations, all under strict privacy and security guarantees. We discuss the details of this approach in Section 5.

KI-4. Data store maintenance under MPC. In Longshot, a data store is an array of encrypted records sorted into ordered bins by domain values, where the number of records in each bin matches the DP synopsis that may not be equal to the true volume of records. When the DP count is larger, the data store is padded with "dummy" records to match the DP count. Additional records are deferred to the next update when the count is lower. There are two challenges to constructing data stores efficiently and privately. First, the untrusted servers generate dummy records for each bin, but generating the exact needed dummy records allows them to learn private information (the true counts of each bin). Thus, we need to add a publicly known number of dummy data records and perform MPC computations. The number needs to be large enough; however, if it is too large, it results in inefficiency. Our solution uses the DP synopsis to estimate the needed dummy records in each bin.

Second, when new data records are added, to avoid privacy leakage in maintaining the data store, the natural solution may require performing MPC computation (specifically, oblivious sorts over real and dummy records) over the entire database, leading to inefficiency. Our solution uses the DP synopsis to publicly identify offsets in each bin to separate mostly real and mostly dummy records. This separation allows us to exclude a vast majority of existing records when performing MPC computation, dramatically improving performance with no additional privacy loss and little effect on accuracy.

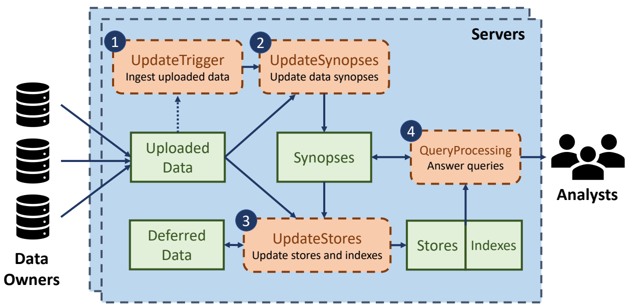


Figure 1: System Diagram.

2.2 Components

Trust model. As seen in Figure 1, we consider three parties: 1) one or more mutually distrustful data owners, 2) the computing servers, and 3) the analysts. Data owners hold database instances that share the same public schema but do not allow each other to view their private data. Each data owner continuously uploads their own private data to the computing servers. The servers store received data, execute pre-specified protocols and process queries sent by analysts, all without seeing private data in the clear. There are two types of analysts, trusted and untrusted, whose roles are agreed upon by all parties before any computation occurs. Untrusted analysts can only issue aggregate queries and are not allowed to see private data in the clear. Trusted analysts can issue non-aggregate queries and directly receive private data from the server, which the analysts can view in the clear. The trusted analysts can be in the same authority as the data owners or sign legal binding as researchers so that they can query the private data directly. All untrusted parties act in a semi-honest fashion, meaning that they faithfully follow the protocol, but may attempt to learn private information.

Secure growing data. Each data owner holds private data records that they upload to the servers for computation. Each record may contain multiple attributes of any data type. Attributes may be either queryable or non-queryable, indicating if they can be computed on as part of a query. There is no restriction on the data type of queryable or non-queryable attributes. To ensure privacy from servers, data owners locally generate *secret shares* [9] of their records and upload different shares to two semi-honest and non-colluding servers; the two jointly compute on secret-shared data using MPC-based protocols. We assume that the number of uploaded records is fixed and known. In practice, this information leak information of private data; we can use techniques from prior works [75] to address this privacy concern w.r.t. upload volume. The Longshot framework supports alternative implementations such as using homomorphic encryption or trusted hardware.

Data structures. On the servers, Longshot uses the following data structures to collect statistics and store the uploaded records over multiple updates: 1) DP synopses, 2) indexes, and 3) data stores. Each materialized DP synopsis is a histogram of a queryable attribute, i.e., a list of noisy counts of records for each domain value, computed over certain blocks of the growing data. Longshot discretizes and encodes the domain of queryable attributes into bins, e.g., values 1-10 are encoded into bin 1 while values 11-20 are encoded into bin 2. Note that the original record is not altered during construction. Once built, the synopsis is used to directly answer aggregate queries and maintain a corresponding index and store. Each index is a CDF version of the corresponding DP synopsis. Each materialized data store generated using the corresponding synopsis contains the aforementioned data blocks ordered by domain values, and the number of records for each domain value maps to that in the synopsis. Thus, the index can locate records in data store to answer arbitrary predicate queries on the domain.

Data buffers. Longshot maintains two data buffers, one for holding newly uploaded data and the other for records deferred during previous updates. Newly uploaded data is appended to the new data buffer. Once an update is triggered, Longshot uses the records in the new data buffer to update the data structures and stored them in the data store, leaving the buffer empty. Any unfetched records are moved to the deferred data buffer for use in the subsequent updates. Details about deferred data are explained in Section 5.3.

2.3 Workflow

Figure 1 showcases the workflow of Longshot where analysts are interested in predicate queries over growing data with a specified data schema from a group of users. Longshot operates in four steps.

Step 1: UpdateTrigger: Data owners encode, secret share and periodically upload their data to servers, which store them in a data buffer. To conceal the size of newly uploaded records, a fixed-size array, including dummy records, is assumed to be uploaded by each data owner at predetermined intervals. The servers use UpdateTrigger at each time step to decide whether to update the data structures with the newly uploaded data.

Step 2: UpdateSynopses. Once UpdateTrigger decides to update, the servers generate new synopses over the newly uploaded records.

Step 3: UpdateStores. Longshot uses the DP synopses, deferred data (not fetched from previous updates), and newly uploaded records to update the index and data store.

Step 4: QueryProcessing. Independently, analysts issue queries to the servers. The servers use the DP synopsis to process aggregate queries and the index and store to process non-aggregate queries. Aligning the index and data store to the DP synopsis ensures that these queries can be answered in the clear (i.e., without using MPC).

3 BACKGROUND

3.1 Notations

Secret shares. We use $\langle x \rangle$ to denote a secret-shared variable x. When x is an array of secret-shared elements, we use $\langle x \rangle[i : j]$ to denote a slice of elements. We denote the function that is computed on secret shares using a secure computation protocol as $\mathcal{F}_{\mathsf{function}}$.

Growing database. Consider a relation with d attributes $R = (att_1, ..., att_d)$. For attribute att_i, we denote its domain by $dom(att_i)$. For the d attributes, we demote the full domain by $\Gamma = dom(att_1) \times ... \times dom(att_d)$. Let D be a database instance, which is a multi-set of elements from Γ and $|D| \geq 0$. Let $|dom(att)| = m$ and we represent the histogram of attribute att as $H \in \mathbb{R}^m$ over the ordered domain $dom(att)$. A histogram H is a list of counts, in which each entry corresponds to a domain value and reports the frequency of the domain value in D. We assume that our model supports queries on a single attribute att for demonstration purposes and discuss the model to support queries on multiple attributes in the Section 7. A growing database consists of a set of insertion or append only logical updates and is a collection of database instances associated with a timestamp. We denote a growing database as $\mathcal{D} = \{D_t\}_{t\geq 1}$, where D_t is a database instance uploaded at time t and $|D_t| \geq 0$.

DP structure. We use Synopsis, $\langle \mathsf{Store} \rangle$ and Index to denote the DP synopses, data stores and indexes, respectively. We define Synopsis as $\{\mathsf{synopsis}_{[i,j]}\}_{1\leq i\leq j}$, where $\mathsf{synopsis}_{[i,j]}$ is a DP histogram with m bins computed by adding DP noise on the true histogram of the queryable attribute att computed on the union of each data block with an update number c $\in [i, j]$ (which will be introduced in Section 5). We define Index = $\{\mathsf{index}_{[i,j]}\}$, where $\mathsf{index}_{[i,j]}$ is a CDF of the histogram $\mathsf{synopsis}_{[i,j]}$. We define $\langle \mathsf{Store} \rangle = \{\langle \mathsf{store}_{[i,j]} \rangle\}$, where $\langle \mathsf{store}_{[i,j]} \rangle$ is a list of secret-shared tuples of the aforementioned data blocks sorted by ordered domains of the queryable attribute att and indexed by $\mathsf{index}_{[i,j]}$. Let $\oplus$ and $\ominus$ denote the bin-wise sum and difference of histograms or CDFs, respectively.

Query. Longshot answers both aggregate and non-aggregate adhoc predicate queries. A query is associated with a predicate $\phi : dom(att) \rightarrow \{0, 1\}$. Evaluating a non-aggregate predicate query q returns all records in $\mathcal{D}$ that satisfy the corresponding predicate ϕ_q, i.e., $q(\mathcal{D}) = \{r \in \mathcal{D} | \phi_q(r.att) = 1\}$. An aggregate version of query q returns $|q(\mathcal{D})|$.

We define a non-aggregate query on indexes and data stores as $\tilde{q}(\langle \mathsf{Store} \rangle, \mathsf{Index})$ and aggregate query results on DP synopsis as $\tilde{q}(\mathsf{Synopsis})$. $\tilde{q}(\langle \mathsf{Store} \rangle, \mathsf{Index})$ includes dummy records to hide the exact volume of query results. We use $|\bar{q}(\langle \mathsf{Store} \rangle, \mathsf{Index})|$ to denote the number of true records returned from $\tilde{q}(\langle \mathsf{Store} \rangle, \mathsf{Index})$. We define the error of an aggregate query $\mathsf{DPCountError}_q$ using $||\tilde{q}(\mathsf{Synopsis}) - |q(\mathcal{D})|\,||_1$. We define the true record error of a non-aggregate query $\mathsf{TrueRecordError}_q$ using $||\ |\bar{q}(\langle \mathsf{Store} \rangle, \mathsf{Index})| - |q(\mathcal{D})|\ ||_1$, which is the number of deferred real data.

3.2 Preliminaries

Secure multi-party computation (MPC). MPC utilizes cryptographic primitives to let multiple parties P jointly compute a function f over the private inputs of each party P_i, without the need for a trusted third party. MPC guarantees that for a probabilistic polynomial time adversary, each party only learns the output of f and nothing about the private data of other parties.

Definition 3.1 ((n, t)-secret sharing). Given a ring $\mathbb{Z}_k$ where $k = 2^\ell$ and ℓ is the length of a secret value x. A (n, t)-secret sharing (t-out-of-n) over $\mathbb{Z}_k$ shares $x \in \mathbb{Z}_k$ with n parties such that at least t parties are required to reconstruct x.

For Longshot, we utilize a $(2,2)$-secret sharing scheme, where each server holds one of two secret shares for a record, and both shares are required in order to reconstruct the plaintext record.

4 PRIVACY MODEL

Definition 4.1 (DP FOR GROWING DATABASES). We define two growing databases $\mathcal{D}$ and $\mathcal{D}'$ as neighboring if they differ by the removal or addition of a single record in the entire collection of uploaded database instances. Let F be a mechanism over a growing database $\mathcal{D}$ and F is ϵ-*DP* if for any $\mathcal{D}$ and $\mathcal{D}'$, and any $O \in \mathcal{O}$, where $\mathcal{O}$ is the output space of F,

$$\Pr[F(\mathcal{D}) \in O] \leq e^{\epsilon} \Pr[F(\mathcal{D}') \in O]$$

We assume that each record is an event (addition of a tuple to the growing database) and thus the mechanism F ensures event-level [30] ϵ-*DP*.

Definition 4.2 (ϵ-SIM-CDP VIEW). A DP structure update protocol Π is said to satisfy ϵ-*SIM-CDP* if for any growing database $\mathcal{D}$ there exists a probabilistic polynomial time (p.p.t.) simulator S, such that for any p.p.t. adversary $\mathcal{A}$, it holds that $VIEW^{\Pi}$ and $VIEW^{S}$ is computationally indistinguishable ($\equiv_c$):

$$\Pr[\mathcal{A}(View^{\Pi}(\mathcal{D}, pp, \kappa)) = 1]$$

$$\equiv_c \Pr[\mathcal{A}(View^{S}(\mathcal{D}, F(\mathcal{D}), pp, \kappa) = 1]$$

where $View^{\Pi}$ denotes the adversary $\mathcal{A}$'s view during the protocol execution which includes a transcript of secret-shared inputs of $\mathcal{D}$ (for one party), randomness, public parameters, messages exchanged and the output of the honest party, and $View^{S}$ denotes the adversary $\mathcal{A}$'s view against the output of the honest party and the simulator S that accesses secret-shared inputs of $\mathcal{D}$ (for one party), the output of F that is ϵ-*DP* and a set of public parameters pp. κ is a computational security parameter.

Relations to other definitions. Using the group privacy property of differential privacy one can easily show that any mechanism satisfying ϵ-event DP also satisfies w-window-event [44] DP with parameter $w\epsilon$ and user level-DP with parameter with privacy loss of $k\epsilon$, where k is an upper bound on the number of tuples contributed by the same user. If there is no finite bound k on user contributions, then one would need to ensure that the algorithms limit the contributions of user in some way. Recent work [17] has shown that attackers could use temporal correlations between events to make the privacy guarantees worse. Nevertheless, they show that for an ϵ-event-DP algorithm, there are constants $\epsilon \leq \alpha, \beta \leq k\epsilon$ such that the privacy loss does not degrade worse than α and β under event- and w-window-event DP respectively. The user level-DP guarantee continues to be $k\epsilon$ even if attackers could use temporal correlations.

Definition 4.2 differs from the standard definition of semi-honest secure computation by allowing leakage in the ideal world to represent what a p.p.t adversary can learn from the execution of protocol Π in the real world. The leakage profile provides differentially private guarantees on input data using the output of DP mechanism F. Thus, the definition ensures differentially private protection against leakage from intermediate/output volume, access patterns and update patterns during protocol execution.

5 PROTOCOL DESIGN

Longshot performs secure computation across two non-colluding and semi-honest servers to update differentially private data structures on uploaded secret-shared data. We begin by looking at how records are ingested by the servers according to an update trigger and how Longshot organizes update numbers of data blocks using a hierarchical time tree. Next, we describe oblivious algorithms to update and maintain the Longshot data structures (Synopsis, Index, and ⟨Store⟩) as new records arrive. Finally, we explain how Longshot utilizes these structures to process and answer ad-hoc predicate queries.

5.1 Longshot Protocol Overview

As described in Section 2.2, Longshot maintains the Synopsis, Index, and ⟨Store⟩ data structures by continually incorporating newly uploaded data. Using a trigger condition, Longshot batches new records together before updating the data structures. We describe the overarching update protocol in Algorithm 1.

Algorithm 1 Longshot Protocol Overview

```
1:  Input: privacy budget $\epsilon$,
2:  Update: Synopsis, ⟨Store⟩, Index
3:  Initialize: c ← 1, ⟨newData_c⟩ ← $\emptyset$, ⟨defer_c⟩ ← $\emptyset$
4:  for $t$ ← 1, ... do
5:      $\langle D_t \rangle$ ← secret-shared records uploaded at time $t$
6:      ⟨newData_c⟩ ← ⟨newData_c⟩ ∪ $\langle D_t \rangle$
7:      res ← UpdateTrigger($t$, ⟨newData_c⟩)
8:      if res == True then
9:          Synopsis ← UpdateSynopses($\epsilon$, ⟨newData_c⟩)
10:         ⟨Store⟩, Index, ⟨defer_{c+1}⟩ ← UpdateStores_OPT(
                Synopsis, ⟨newData_c⟩, ⟨defer_c⟩)
11:         c ← c + 1, ⟨newData_c⟩ ← $\emptyset$
```

The input to Algorithm 1 is the privacy budget ϵ provided to Longshot as a parameter. At timestamp t, Longshot stores newly uploaded data D_t in a new data buffer $\langle \mathsf{newData_c} \rangle$ where c refers to the current update number (Alg 1: 5). Then, Longshot determines whether to update the data structures according to the provided UpdateTrigger policy (Alg 1: 7). Once an update is triggered, Longshot updates Synopsis using $\langle \mathsf{newData_c} \rangle$ and privacy budget ϵ (Alg 1: 9). Next, Longshot uses Synopsis, deferred data $\langle \mathsf{defer_c} \rangle$ from the previous update and $\langle \mathsf{newData_c} \rangle$, to update Index, ⟨Store⟩ and $\langle \mathsf{defer_{c+1}} \rangle$ (Alg 1: 10). Finally, Longshot empties $\langle \mathsf{newData_c} \rangle$ and increments the update number to create $\langle \mathsf{newData_{c+1}} \rangle$ (Alg 1: 11). We describe the details of the algorithm below.

Update trigger. The UpdateTrigger can fire either at fixed time intervals or based on the volume of uploaded data records. Previous work [51, 75, 76] considers how to trigger updates to handle sparse data based on upload volume while preserving privacy; their techniques are complementary to ours and can be applied directly.

Hierarchical update tree. The update trigger separates uploaded records into disjoint blocks of data where each block is associated with an update number c. Naïvely, we could use the privacy budget to generate a DP histogram for the data block of each update. However, this leads to a linearly increasing total error in the number of

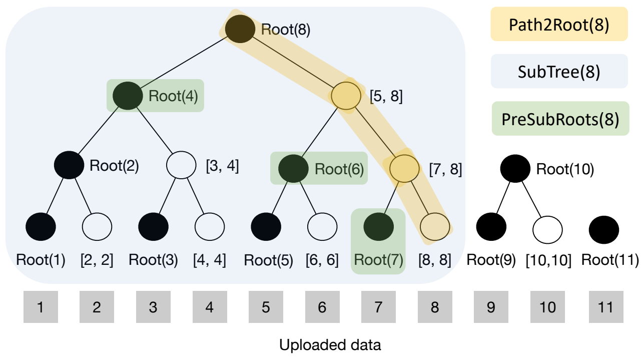


Figure 2: Hierarchical tree of updates.

updates over time. Longshot instead arranges updates in a tree of hierarchical intervals as shown in Figure 2 to achieve logarithmic error over time. Formally, in a tree of intervals with branching factor b, each leaf node of the tree is a unit-length interval while non-leaf nodes correspond to intervals that are recursively the unions of the intervals of b children nodes. Since the tree is materialized gradually over time, we maintain a set of sub-trees and take a binary tree ($b = 2$) as a running case where h is the height of the highest tree for the maximum number of updates T.

In order to carry out the update protocol, we define four functions to return specific intervals and illustrate them using the 8-th update as an example in Figure 2. Root(i) returns the interval of the root of the sub-tree whose rightmost node is the i-th leaf node. Path2Root(i) returns the set of intervals that are on the path from the i-th leaf node to Root(i). SubTree(i) returns all intervals in the sub-tree of Root(i). PreSubRoots(i) returns the smallest set of intervals that together with the i-th leaf node covers Root(i). To make terminology simple, the data blocks of an interval refers to the union of all data blocks whose update number falls in the interval; the synopses, indexes and stores of an interval refers to the data structures generated using the data blocks of that interval.

For each update i, we generate the data structures of Root(i). We call this update strategy the *tree approach*. Looking ahead, the DP synopsis generated at each update would use the total privacy budget divided by the tree height, which is logarithmic in the number of updates. A logarithmic number, rather than a linear number of synopses, can cover all uploaded data so that the tree approach can provide logarithmic variance in the number of updates. The detailed error analysis is in Section 5.5.

5.2 Protocol to Update Synopsis

We now explain the details of how Longshot securely and efficiently generates an accurate synopsis of Root(i) using differential privacy with the tree approach for update number i. Once generated, the synopses can be used to: 1) answer arbitrary predicate aggregate queries (described in Section 5.4) and 2) update indexes and stores (described in Section 5.3). Recall that a DP synopsis represents a histogram over data blocks of records from a specific range of updates. Although we can compute this by applying a standard differentially private mechanism [29] on the data blocks of Root(i), Longshot further boosts accuracy by enforcing consistency on noisy measurements at different hierarchy levels [38, 63]. Longshot computes the DP histograms of SubTree(i), and for each bin, apply constrained inference [38, 63] on the DP counts of SubTree(i) to obtain improved estimates of each count of Root(i). We discuss the details of the UpdateSynopses protocol in Algorithm 2 below.

Algorithm 2 Update DP synopsis

```
    // $\{\langle H_i \rangle\}$: set of secret-shared true histograms,
    // $\{\tilde{H}_{[i,j]}\}$: set of DP histograms
1:  function UpdateSynopses($\epsilon$, $\langle \mathsf{newData_c} \rangle$):
2:      $\langle H_c \rangle \leftarrow \mathcal{F}_{\mathsf{ComputeTrueHist}}(\langle \mathsf{newData_c} \rangle)$
3:      $\{\tilde{H}_I\}_{I \in \mathsf{Path2Root(c)}} \leftarrow \mathcal{F}_{\mathsf{ComputeDPHists}}(\{\langle H_i \rangle\}_{i \in \mathsf{Root(c)}}, \epsilon)$
4:      store $\langle H_c \rangle$ and $\{\tilde{H}_I\}_{I \in \mathsf{Path2Root(c)}}$ internally
5:      $\mathsf{synopsis_{Root(c)}} \leftarrow \mathsf{BoostAccuracy}(\{\tilde{H}_I\}_{I \in \mathsf{SubTree(c)}})$
6:      return Synopsis $\cup$ $\mathsf{synopsis_{Root(c)}}$
    // $m$: number of bins
7:  function $\mathcal{F}_{\mathsf{ComputeDPHists}}(\{\langle H_i \rangle\}_{i \in \mathsf{Root(c)}}, \epsilon)$ :
8:      $\tilde{H}_I \leftarrow (\oplus_{i \in I} \langle H_i \rangle) \oplus \mathcal{F}_{\mathsf{GenLap}}(\frac{\epsilon}{h}, m)$ for $I \in$ Path2Root(c)
9:      return $\{\tilde{H}_I\}_{I \in \mathsf{Path2Root(c)}}$
10: function BoostAccuracy($\{\tilde{H}_I\}_{I \in \mathsf{SubTree(c)}}$)
11:     for each bin $\in [1, m]$ do:
12:         $\mathsf{synopsis_{Root(c)}}$[bin] = WAVG($\{\tilde{H}_I[\mathsf{bin}]\}_{I \in \mathsf{SubTree(c)}}$)
13:     return $\mathsf{synopsis_{Root(c)}}$
```

Step 1: $\mathcal{F}_{\mathsf{ComputeTrueHist}}$. The first step is to securely compute the true histogram of the data block with current update number c and output the secret-shared version $\langle H_c \rangle$. We compute the histogram using the most efficient oblivious algorithm which is a combination of oblivious sorts and linear scans (akin to the gather protocol in GraphSC [60]). This requires $O((n+m)\log^2(n+m))$ runtime, where n and m are the sizes of input data and bins, respectively (Alg 2: 2).

Step 2: $\mathcal{F}_{\mathsf{ComputeDPHists}}$. We compute the DP histograms of the data blocks with update intervals of Path2Root(c), so those of SubTree(c) are computed over each update (Alg 2: 3). $\mathcal{F}_{\mathsf{ComputeDPHists}}$ takes the secret-shared true histogram of each entry in Root(c) as input, sums them together to compute the true histograms of each interval of Path2Root(c), and then adds noise to the true histograms according to Laplace mechanism [29] with privacy budget $\frac{\epsilon}{h}$, where h is the height of the tree corresponding to the maximum number of updates. $\mathcal{F}_{\mathsf{ComputeDPHists}}$ reveals these noisy histograms in the clear. We implement $\mathcal{F}_{\mathsf{GenLap}}(\frac{\epsilon}{h}, m)$ to generate a vector of m Laplace noise values with scale $\frac{\epsilon}{h}$ using the approach proposed in Cryptϵ [21], where for each value, two servers sample independently with scale $\frac{\epsilon}{h}$ and add them up using MPC (Alg 2: 7-9). An alternative approach is for servers to jointly sample noise with scale $\frac{\epsilon}{h}$ under MPC (as in [22, 76]).

Step 3: BoostAccuracy. To obtain an improved estimation of the DP histogram of Root(c), we apply the weighted average step of constrained inference described by Hay et al. [38, 63] on the tree bin-wise (Alg 2: 5). The precise definition of weighted average (WAVG) is in Hay et al. [38, Section 4.1]. As mentioned earlier, enforcing consistency improves the accuracy of $\mathsf{synopsis_{Root(c)}}$ without additional privacy loss (Alg 2: 10-13).

5.3 Protocol to Update Index and ⟨Store⟩

After generating a Synopsis for an array of uploaded records, Longshot creates two corresponding data structures, Index and ⟨Store⟩, to lookup and store those records. Thus, the servers can efficiently lookup the locations of records in cleartext for non-aggregate queries using the Index, and fetch them from the ⟨Store⟩ (described in detail in Section 5.4). In order to store uploaded records correctly, we introduce the $\mathcal{F}_{\text{ThresholdedBinSort}}$ primitive, seen in Figure 3. This primitive takes in an array of secret shared records $\langle V\rangle$ for whom we have already generated a histogram and obliviously sorts them into bins such that the bin sizes match the histogram. We describe the details below in Algorithm 3.

Algorithm 3 $\mathcal{F}_{\text{ThresholdedBinSort}}$ primitive

```
Input: ⟨V⟩: array where each element has the form (data, bin, isReal) and bin ∈ [1, m], $\tilde{H}$: an array of m counts
1:  function $\mathcal{F}_{\text{ThresholdedBinSort}}(\langle V\rangle, \tilde{H})$
        // A: list of tuples [data, bin, isCounter, isReal, mark]
2:      $\langle A\rangle \leftarrow$ transform($\langle V\rangle, \tilde{H}$)
        ▷ Step 1: Group by bins
3:      sort $\langle A\rangle$ by (bin, ¬isCounter, ¬isReal)
        ▷ Step 2: Mark whether records are in ⟨output⟩ or ⟨defer⟩
4:      var cnt := 0
5:      for $i \leftarrow 1, ..., |A|$ do
6:          if $\langle A\rangle[i]$.isCounter then
7:              cnt = $\langle A\rangle[i]$.mark, $\langle A\rangle[i]$.mark = $|\tilde{H}| + 2$
8:          else
9:              cnt = cnt - 1
10:             if cnt < 0 then $\langle A\rangle[i]$.mark = $|\tilde{H}| + 1$
        ▷ Step 3: Generate outputs
11:     sort $\langle A\rangle$ by (mark, ¬isReal)
12:     $k \leftarrow \sum_{i\in[1,|\tilde{H}|]} \tilde{H}[i]$
13:     ⟨output⟩ $\leftarrow \langle A\rangle[1:k]$, ⟨defer⟩ $\leftarrow \langle A\rangle[k+1 : |A| - |\tilde{H}|]$
14:     return ⟨output⟩, ⟨defer⟩
```

We first transform the array $\langle V\rangle$ and threshold counts $\tilde{H}$ into an alternative representation $\langle A\rangle$ using a list of tuples tagged with [data, bin, isCounter, isReal, mark]. First, we set isCounter values for the input array as 0 and mark values as their bin values. Note that data, bin, isReal represent non-queryable attributes, queryable attribute and dummy/real for a record in input array. Second, we append m secret-shared records to the input array. The bin values for the m records are from 1 to m and the mark values are initialized to the appropriate values in $\tilde{H}$. isCounter and isReal values for the m records are 1 and 0 respectively. This is described as the transform() function in Algorithm 3 line 2 for simplicity. We provide an example of three bins to illustrate the three steps of $\mathcal{F}_{\text{ThresholdedBinSort}}$ in Figure 3. The algorithm progresses in three steps.

Step 1: Group by bins. $\langle A\rangle$ is obliviously sorted by (bin, isCounter, isReal) to group records of the same bin together. isCounter record is followed by real and dummy records within each bin. (Alg 3: 3).

Step 2: Mark whether records are in ⟨output⟩ or ⟨defer⟩. Our eventual goal is to have an output ⟨output⟩ with bins of pre-specified sizes containing preferably real elements. Thus, the objective is to decide for each bin if an element would be in the ⟨output⟩ or ⟨defer⟩. Specifically, based on the mark values when isCounter is true for a record (i.e., records coming from $\tilde{H}$), we keep the mark values (in the output) or assign $m+1$ as the mark values (in the deferred category) for other elements. We assign $m+2$ as the mark values for counter records to move them to the end. (Alg 3: 4-10).

Step 3: Generate outputs. We sort the array by mark values. The first $\sum_i \tilde{H}[i]$ elements are output as ⟨output⟩, the last $|\tilde{H}|$ elements are ignored, and the remaining elements form ⟨defer⟩. Within ⟨output⟩ and ⟨defer⟩, real records are before dummy records (Alg 3: 11-14).

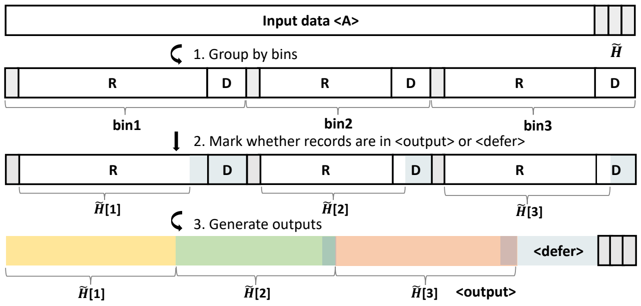


Figure 3: $\mathcal{F}_{\text{ThresholdedBinSort}}$: R: Real; D: Dummy. Each color in the last row represents each bin. The shaded part in the fourth row represents dummy records.

Updating Index and ⟨Store⟩. For a given update c, we have shown how to create synopsis$_{\text{Root(c)}}$ for the records uploaded in c. Now, we want to store those records in ⟨store$_{\text{Root(c)}}$⟩ and lookup them up using index$_{\text{Root(c)}}$. There are two high-level challenges to generate ⟨store$_{\text{Root(c)}}$⟩. First, updating the store obliviously can lead to accessing the entire database(e.g., when generating ⟨store$_{\text{Root(8)}}$⟩), significantly reducing performance. Second, Longshot allocates space for storing records based on the synopsis, which incorporates positive or negative Laplace noise in each bin and may not accurately reflect the exact number of records. Longshot handles excess space by inserting *dummy* records, while for inadequate space, additional records are *deferred.* We introduce UpdateStores$_{\text{ALL}}$ and UpdateStores$_{\text{OPT}}$ to solve the second and the first challenge.

5.3.1 UpdateStores$_{\text{ALL}}$ *Protocol.*

Let's start with a high-level description of UpdateStores$_{\text{ALL}}$. At update c, we have records uploaded during previous updates (located in either ⟨defer$_c$⟩) or stores, newly uploaded records, and dummy records (added to hide the data distribution from the untrusted servers). We concatenate and sort these records using $\mathcal{F}_{\text{ThresholdedBinSort}}$, constructing ⟨store$_{\text{Root(c)}}$⟩ using most of the records and moving the rest into ⟨defer$_{c+1}$⟩. We discuss the details of this process in Algorithm 4.

Step 1: Add dummy records. As the synopsis is noisy, it may require adding dummy records to the bins in the data store. Therefore, we commence the algorithm by adding an adequate number of dummy records. Adding as many dummy records as in synopsis$_{\text{Root(c)}}$ for each bin is sufficient but inefficient. However, an overly small size leads to inconsistency between ⟨store$_{\text{Root(c)}}$⟩ and synopsis$_{\text{Root(c)}}$.

We use insight to choosing the appropriate size: DP noise added in synopsis$_{\text{Root(c)}}$ determines the number of dummy records in ⟨store$_{\text{Root(c)}}$⟩. If we can bound the DP noise with high probability, then we can bound the number of dummies needed in ⟨store$_{\text{Root(c)}}$⟩

Algorithm 4 ⟨Store⟩ and Index update on ALL

```
    // $m$: number of bins, $d$: DP noise bound
1:  function UpdateStores$_{\mathsf{ALL}}$(Synopsis, ⟨newData$_c$⟩, ⟨defer$_c$⟩):
2:      $\langle R_{\mathrm{pre}}\rangle \leftarrow \{\langle \mathrm{store}_I\rangle\}_{I\in \mathrm{PreSubRoots(c)}}$
3:      $\tilde{H}_{\mathrm{Root(c)}} \leftarrow \mathrm{synopsis}_{\mathrm{Root(c)}}$
4:      ⟨store$_{\mathrm{Root(c)}}$⟩, ⟨defer$_{c+1}$⟩ ← UpdateStores(⟨newData$_c$⟩,
            ⟨defer$_c$⟩, $\langle R_{\mathrm{pre}}\rangle, \tilde{H}_{\mathrm{Root(c)}}$)
5:      return updated ⟨Store⟩, Index, ⟨defer$_{c+1}$⟩
6:  function UpdateStores(⟨newData$_c$⟩, ⟨defer$_c$⟩, $\langle R_{\mathrm{pre}}\rangle, \tilde{H}_{\mathrm{sort}}$):
        ▷ Step 1: Add dummy records
7:      ⟨dummy⟩ ← Generate $m * d$ dummy records
        ▷ Step 2: $\mathcal{F}_{\mathrm{ThresholdedBinSort}}$
8:      $\langle V\rangle \leftarrow \langle R_{\mathrm{pre}}\rangle$ || ⟨defer$_c$⟩ || ⟨newData$_c$⟩ || ⟨dummy⟩
9:      $\tilde{H}$ ← Process $\tilde{H}_{\mathrm{sort}}$ into $\tilde{H}$ such that $\sum_i \tilde{H}[i] \le |\langle V\rangle|$
10:     ⟨output⟩, ⟨defer⟩ ← $\mathcal{F}_{\mathrm{ThresholdedBinSort}}(\langle V\rangle, \tilde{H})$
        ▷ Step 3: Truncate dummy records in ⟨defer⟩
11:     $C_{\mathrm{cut}} \leftarrow m \times d \times (|\mathrm{PreSubRoots(c)}|)$
12:     ⟨defer$_{c+1}$⟩ ← ⟨defer⟩$[1 : |\langle\mathrm{defer}\rangle| - C_{\mathrm{cut}}]$
13:     return ⟨output⟩, ⟨defer$_{c+1}$⟩
```

using a fixed value. We can use this value to generate enough dummy records instead of using the maximum value. In particular, based on the fact below, the DP noise, sampled from Laplace distribution with scale b, is bounded by $d = \ln(\frac{1}{p}) \cdot b$ with probability $1 - p$, where $p = exp(-t)$. We securely generate d dummy records for each bin value as ⟨dummy⟩. Therefore, there are enough dummy records for each bin for $\mathcal{F}_{\mathrm{ThresholdedBinSort}}$ to update the data store with probability $1 - p$. We deal with the scenario where we do not have sufficiently many dummies in step 2 below (Alg 4: 7).

$$\mathrm{Fact1} : If\ \mathrm{Y} \sim \mathrm{Lap(b)},\ then:\ \Pr[|\mathrm{Y}| \ge \mathrm{t} \times \mathrm{b}] = \exp(-\mathrm{t})$$

Step 2: $\mathcal{F}_{\mathrm{ThresholdedBinSort}}$. Now that we have enough dummies with high probability, Longshot calls the $\mathcal{F}_{\mathrm{ThresholdedBinSort}}$ primitive to process the input array ($\{\langle \mathrm{store}_I\rangle\}_{I\in\mathrm{PreSubRoots(c)}}$ || ⟨defer$_c$⟩ || ⟨newData$_c$⟩ || ⟨dummy⟩) according to synopsis$_{\mathrm{Root(c)}}$ to obtain ⟨store$_{\mathrm{Root(c)}}$⟩ and update ⟨defer$_{c+1}$⟩. In case added dummy records are not sufficient, we pre-process synopsis$_{\mathrm{Root(c)}}$ to obtain a histogram $\tilde{H}$ so that $\sum_i \tilde{H}[i]$ is equal to the length of the input array. We processes each value of the cumulative distribution function (CDF) of synopsis$_{\mathrm{Root(c)}}$ to be at most the length of input array and then compute the histogram $\tilde{H}$ from the processed CDF (Alg 4: 8-10).
Step 3: Truncate dummy records in ⟨defer⟩. The errors in DP synopsis accumulates logarithmically in the number of updates as mentioned in Section 5.1. Correspondingly, this holds true for the number of dummy records accumulated in the deferred data buffer. We achieve the logarithmically accumulated dummies by dropping the last $m \times d \times |\mathrm{PreSubRoots(c)}|$ records in ⟨defer⟩ (Alg 4: 11-12).

5.3.2 UpdateStores$_{\mathsf{OPT}}$ Protocol. With UpdateStores$_{\mathsf{ALL}}$, Longshot must obliviously sort all data blocks of Root(c). When the update number c is a power of two, we end up obliviously sorting the entire data store, which can be extremely inefficient. To address this concern, we introduce UpdateStores$_{\mathsf{OPT}}$, which significantly reduces the number of records that Longshot needs to sort using expensive cryptographic protocols.

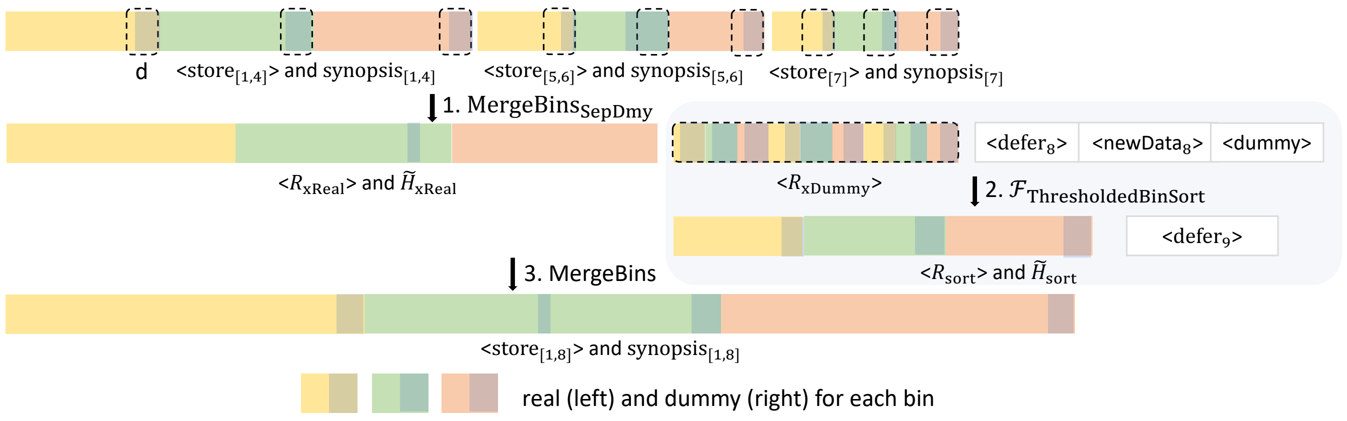
In UpdateStores$_{\mathsf{OPT}}$, we extend the insight w.r.t. bounding the DP noise used in synopsis$_{\mathrm{Root(c)}}$ to sorting records. Since real records are stored before dummy records in each bin, we have a region where the records are all dummies with high probability. We can apply ThresholdedBinSort only on the records in this region in $\{\langle \mathrm{store}_I\rangle\}_{I\in\mathrm{PreSubRoots(c)}}$, allowing to either replace extra dummy records with real records or move them into ⟨defer⟩.

We provide examples using three bins at the 8-th update in Figure 4 to illustrate how UpdateStores$_{\mathsf{OPT}}$ applies $\mathcal{F}_{\mathrm{ThresholdedBinSort}}$ only on a small portion of the input array, rather than all input data as in UpdateStores$_{\mathsf{ALL}}$. We describe the protocol in Algorithm 5.

Algorithm 5 Optimized ⟨Store⟩ and Index update

```
    // $m$: number of bins, $d$: DP noise bound
1:  function UpdateStores$_{\mathsf{OPT}}$(Synopsis, ⟨newData$_c$⟩, ⟨defer$_c$⟩) :
        ▷ Step 1: MergeBins$_{\mathsf{SepDmy}}$
2:      $(\langle R_{\mathrm{xReal}}\rangle, \tilde{H}_{\mathrm{xReal}}), \langle R_{\mathrm{xDummy}}\rangle$ ← MergeBins$_{\mathsf{SepDmy}}$
            ($d$, synopsis$_I$, ⟨store$_I$⟩)$_{I\in\mathrm{PreSubRoots(c)}}$)
        ▷ Step 2: Call UpdateStores on mostly dummy elements.
3:      $\tilde{H}_{\mathrm{sort}} \leftarrow \mathrm{synopsis}_{\mathrm{Root(c)}} \ominus \tilde{H}_{\mathrm{xReal}}$
4:      $\langle R_{\mathrm{sort}}\rangle$, ⟨defer$_{c+1}$⟩ ← UpdateStores(⟨newData$_c$⟩,
            ⟨defer$_c$⟩, $\langle R_{\mathrm{xDummy}}\rangle, \tilde{H}_{\mathrm{sort}}$)
        ▷ Step 3: MergeBins
5:      ⟨store$_{\mathrm{Root(c)}}$⟩ ← MergeBins($(\langle R_{\mathrm{sort}}\rangle, \tilde{H}_{\mathrm{sort}})$,
            $(\langle R_{\mathrm{xReal}}\rangle, \tilde{H}_{\mathrm{xReal}})$)
6:      return updated ⟨Store⟩, Index, ⟨defer$_{c+1}$⟩
7:  function MergeBins$_{\mathsf{SepDmy}}$($d$, (synopsis$_I$,
        ⟨store$_I$⟩)$_{I\in\mathrm{PreSubRoots(c)}}$) :
8:      for each bin $i \in [1, m]$ do:
9:          for $I \in$ PreSubRoots(c) do:
10:             $R$ ← records for bin $i$ in ⟨store$_I$⟩ located by DP index
11:             $C_r \leftarrow \max(0, (\mathrm{synopsis}_I[i] - d))$
12:             $\tilde{H}_{\mathrm{xReal}}[i] \leftarrow \tilde{H}_{\mathrm{xReal}}[i] + C_r$
13:             $R_{\mathrm{xReal}} \leftarrow R_{\mathrm{xReal}} \,||\, R[1 : C_r]$
14:             $R_{\mathrm{xDummy}} \leftarrow R_{\mathrm{xDummy}} \,||\, R[C_r + 1, \mathrm{synopsis}_I[i]]$
15:     return $(\langle R_{\mathrm{xReal}}\rangle, \tilde{H}_{\mathrm{xReal}}), \langle R_{\mathrm{xDummy}}\rangle$
```

Step 1: MergeBins$_{\mathsf{SepDmy}}$. Given d, which bounds the number of dummies in each bin, MergeBins$_{\mathsf{SepDmy}}$ treats the last d records from each bin in $\{\mathrm{store}_I\}_{I\in\mathrm{PreSubRoots(c)}}$ as containing most dummies and moves them into $\langle R_{\mathrm{xDummy}}\rangle$. If the number of records for a bin in $\{\mathrm{synopsis}_I\}_{I\in\mathrm{PreSubRoots(c)}}$ is smaller than d, all records in the bin are treated as dummies and moved to $\langle R_{\mathrm{xDummy}}\rangle$. Next, MergeBins$_{\mathsf{SepDmy}}$ merges bin-wise the remaining records, which are mostly real, into $\langle R_{\mathrm{xReal}}\rangle$ and computes the DP histogram $\tilde{H}_{\mathrm{xReal}}$ of $\langle R_{\mathrm{xReal}}\rangle$. (Alg 5: 2). Note that, these operations are based on public parameters, and thus they can be performed in the clear.
Step 2: Call UpdateStores *on mostly dummy elements.* We now apply UpdateStores on all records of $\langle R_{\mathrm{xDummy}}\rangle$ together with ⟨defer$_c$⟩, ⟨newData$_c$⟩, and ⟨dummy⟩. At the end of this process, we obtain a sorted array $\langle R_{\mathrm{sort}}\rangle$ which is consistent with $\tilde{H}_{\mathrm{sort}}$ and $\langle R_{\mathrm{sort}}\rangle$ contains as many real records as possible from $\langle R_{\mathrm{xDummy}}\rangle$, ⟨defer$_c$⟩ and ⟨newData$_c$⟩. This ensures that the merged array ⟨store$_{\mathrm{Root(c)}}$⟩ from $\langle R_{\mathrm{sort}}\rangle$ and $\langle R_{\mathrm{xReal}}\rangle$ in step 3 is consistent with synopsis$_{\mathrm{Root(c)}}$ and contains mostly real records (Alg 5: 3-4).



d
<store[1,4]> and synopsis[1,4]
<store[5,6]> and synopsis[5,6]
<store[7]> and synopsis[7]
1. MergeBinsSepDmy
<RxReal> and H̃xReal
<RxDummy>
<defer8>
<newData8>
<dummy>
2. ℱThresholdedBinSort
<defer9>
<Rsort> and H̃sort
3. MergeBins
<store[1,8]> and synopsis[1,8]
real (left) and dummy (right) for each bin

Figure 4: $\mathsf{UpdateStores}_{\mathsf{OPT}}$**: The example uses the DP stores of intervals** $[1,4]$, $[5,6]$, $[7,7]$ **returned by** $\mathsf{PreSubRoots}(8)$ **and the data block** $[8,8]$ **stored in the new data buffer** ρ_8 **to update the DP store of** $[1,8]$**. Each color represents the location of each bin in these stores identified by their index, and the dotted box represents the records seen as dummies by their index. The shaded area represents actual dummy records that are unknown either in public or by index.**

Step 3: MergeBins. We merge the sorted data $\langle R_{\mathsf{sort}}\rangle$ with the mostly real array $\langle R_{\mathsf{xReal}}\rangle$ bin-wisely using their histograms $\tilde{H}_{\mathsf{sort}}$ and $\tilde{H}_{\mathsf{xReal}}$ to obtain $\langle \mathsf{store}_{\mathsf{Root(c)}}\rangle$. The two arrays are consistent with their DP histograms and thus we can locate the records for each bin without decrypting data (Alg 5: 5).

5.4 QueryProcessing Protocol

We now discuss how Longshot utilizes our data structures to answer ad-hoc predicate queries posed by the analyst. The Synopsis generated with DP guarantees is publicly released, allowing for the direct answering of aggregate queries without the need to fetch individual records. Longshot utilizes the publicly released Index to locate and fetch the needed secret-shared records in $\langle\mathsf{Store}\rangle$ for Non-aggregate predicate queries. We discuss the details below.

Step 1: Map query. To process a query q, we check the update number c at the current time stamp and compute the set of bins for which the predicate of q returns true as $B = \{i \in [1:m] | \phi_q(i) = 1\}$.

Step 2: Prepare data structures. We perform this step only for the first query for a given update number. We merge the data stores of Intervals(c) using the corresponding indexes and sum up these DP synopses and indexes, bin by bin.

Step 3: Process query. For an aggregate query, we use the processed DP synopsis to compute the total count for the required bins B. For a non-aggregate query, we use the processed index and data store to fetch the total records for the required bins B.

5.5 Error and Efficiency Analysis

Query errors. We analyze DPCountError and TrueRecordError for aggregate and non-aggregate queries. We assume that the total number of updates is T. For each bin, the variance of the sum of counts in Synopsis over T updates is $O((\frac{\log T}{\epsilon})^2) \times \log T = O(\frac{\log^3 T}{\epsilon^2})$. The variance decides the DPCountError.

The negative noise added in a synopsis decides the size of delayed real records in the corresponding store, leading to the error of TrueRecordError. However, TrueRecordError is further affected by the size of $\langle\mathsf{dummy}\rangle$ records and the value of d, which is one of the inputs of $\mathsf{MergeBins}_{\mathsf{SepDmy}}$. When there are not enough dummy records in each bin, the number of records for each bin in store is inconsistent with the bin count in the corresponding synopsis. Thus, index can locate the records in store incorrectly, potentially resulting in high TrueRecordError. When d is smaller, fewer records for each bin in $\{\mathsf{store}_I\}_{I\in\mathsf{PreSubRoots(c)}}$ are treated as dummies by $\mathsf{MergeBins}_{\mathsf{SepDmy}}$. Therefore, more dummy records are unsorted and accumulated in $\mathsf{store}_{\mathsf{Root(c)}}$, and thus more real records are deferred, increasing TrueRecordError. Both the size of $\langle\mathsf{dummy}\rangle$ and the value of d are decided by parameter p in the $\mathsf{UpdateStores}_{\mathsf{ALL}}$ and $\mathsf{UpdateStores}_{\mathsf{OPT}}$ protocol.

Efficiency. The update protocol for $\langle\mathsf{Store}\rangle$ primarily computes $\mathcal{F}_{\mathsf{ThresholdedBinSort}}$ using MPC. $\mathcal{F}_{\mathsf{ThresholdedBinSort}}$ includes two oblivious sorts and one linear scan, and thus the protocol takes $O((n+m)\cdot\log^2(n+m))$, where n is the number of records to sort and m is the number of bins. Let n be the sum of the size n_{defer} of real and dummy data acumulated in the deferred data buffer and the size n' of the necessary data to sort for the required data block. $n_{\mathsf{defer}} = O(\sqrt{\log k})$. Assume that the number of uploaded records at each triggered update is N and the height of SubTree(i) is h'. For $\mathsf{UpdateStores}_{\mathsf{OPT}}$, $n' = N + h' \times d \times m$, compared with $N \times 2^{h'}$ for $\mathsf{UpdateStores}_{\mathsf{ALL}}$. The protocol of updating DP synopsis requires $O((n+m)\cdot\log^2(n+m) + m\cdot h')$ runtime, where $n = N$. Longshot processes queries publicly, so the processing time is negligible.

6 EXPERIMENTAL EVALUATION

In this section, we describe the evaluation results of Longshot. Specifically, we address the following questions:

- **Question-1** *On processing ad-hoc predicate queries:*
  - Efficiency: Is Longshot more efficient than baseline approaches?
  - Errors: How does query error for Longshot compare to baseline approaches?

- **Question-2** *On updating the data structures:* Does the optimized UpdateStores$_{\text{OPT}}$ protocol solve the bottleneck of updating data stores in UpdateStores$_{\text{ALL}}$?
- **Question-3** *On tunable trade-offs:* How are privacy, efficiency and accuracy affected when adjusting the privacy budget?

6.1 Methodology

Implementation and configurations. We use C++ and EMP toolkit [1] to implement Longshot with secure 2-PC protocols. The experiments are run on a single machine with 64-bit Ubuntu 20.04.4, 8 cores, 2.50GHz CPU, and 32Gb RAM using two separate processes.

Dataset. We evaluate Longshot on the NYC Taxi dataset [2]. After we dropped the records with missing and NULL values, the NYC Taxi dataset has 1363103 records. We use the 'total_amount' column whose values are numerical, and we bucketize them to obtain $m = 40$ bins. For a large number of updates, we copy over the original dataset to ensure sufficiently many records.

Query. We answer two workloads of 40 point queries and 800 range queries on all uploaded data over time. For each query workload, we report the average DPCountError and TrueRecordError of aggregate and non-aggregate queries and average runtime of aggregate and non-aggregate queries.

Default setting. We set the privacy budget ϵ over all updates to 1, by default. The failure probability of bounding Laplace noise, p, for UpdateStores is set to 0.001. UpdateTrigger uses a public timer to trigger updates and the number N of new records uploaded at each triggered update is 1000. Unless specified explicitly, the largest number of updates T is 200, and the height of the tree is $h = \log_2 T$.

6.2 Baseline Algorithms

In Section 5, we show how to generate Synopsis, and present two protocols UpdateStores$_{\text{ALL}}$ and UpdateStores$_{\text{OPT}}$ that use Synopsis to update the data store. We refer to these two resulting systems as Tree-ALL and Tree-OPT respectively. We compare against two additional protocols described below.

The first protocol, Baseline, processes an ad-hoc query independently on all uploaded data with differentially-private leakage. Baseline extends the approaches seen in IncShrink [76], which requires one query prespecified, to answer ad-hoc queries. Through this comparison, we can see the necessity of the DP structures created in Longshot for efficient ad-hoc query execution. In Baseline, given an aggregate query, we linear scan all uploaded data to compute a differentially private count using the privacy budget. To answer a non-aggregate query, we linear scan all records to mark the required records for the query, obliviously sort all data to move the target records to the beginning, and return the first DP count number of records. Each non-aggregate query takes $O(n \cdot \log^2 n)$ time, where n is the size of all the data uploaded until a given time point. We only evaluate the query errors when an update is triggered. Therefore, at each update in Baseline, we set the privacy budget for the point query as $\frac{\epsilon}{T}$ and for each range query as $\frac{\epsilon}{T \times m}$, where m is the number of bins.

The second protocol, Leaf, uses an alternative to the hierarchical tree approach described in Section 5.1. In particular, the Leaf is a special case of the hierarchical approach where the height h and the branching factor b of the hierarchical time tree is 1. Thus, Root(i), Path2Root(i) and SubTree(i) only return $[i, i]$, PreSubRoots(i) returns empty and Intervals(i) returns $[1, 1], \dots, [i, i]$. Consequently, UpdateStores$_{\text{ALL}}$ and UpdateStores$_{\text{OPT}}$ devolve to the same update protocol. In this protocol, the privacy budget used at each update is the total privacy budget. Therefore, the protocol has the advantage of obtaining a low error when the number of updates is small.

The runtime of updating DP stuctures of Longshot is comparable to the runtime of processing one query using Baseline, however, DP stuctures can be used to process unlimited queries once released.

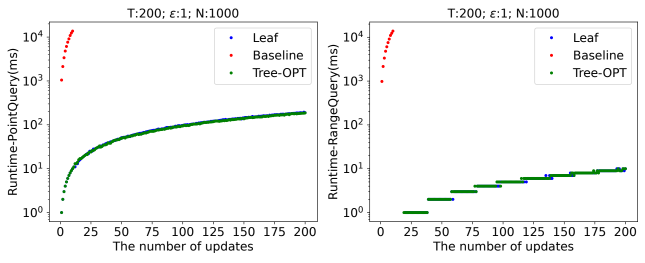
Table 1: Comparison of aggregate query variance and maximum records processed under MPC for query processing and data update between Baseline, Leaf, Tree-OPT **and** Tree-ALL. k **is the number of updates.**

	Agg. error	Query	Update
Baseline	$O(\frac{T^2}{\epsilon^2})$	$O(kN)$	public
Leaf	$O(\frac{T}{\epsilon^2})$	public	$O(N + \sqrt{k} + m)$
Tree-OPT	$O(\frac{\log^3 T}{\epsilon^2})$	public	$O(N + md \log k + \sqrt{\log k} + m)$
Tree-ALL	$O(\frac{\log^3 T}{\epsilon^2})$	public	$O(kN + \sqrt{\log k} + m)$

6.3 End-to-End comparison

We answer Q1 by comparing Leaf, Baseline, and Tree-OPT. Tree-OPT and Tree-ALL operate on similar structures for query processing, and their differences are explained in Figure 8. The blue points, red points, and green points in Figures 5, 6 and 7 represent the results of Leaf, Baseline, and Tree-OPT, respectively.

Observation 1. Leaf **and** Tree-OPT **protocols provide orders of magnitude performance improvement over the** Baseline. Figure 5 shows the run time of processing aggregate and non-aggregate queries on all data at each triggered update. We observe that Baseline is much less efficient than Leaf and Tree-OPT. Despite the possibility of different dummy records in DP store, the query processing runtimes for Leaf and Tree-OPT are almost the same as when performed in clear text. The processing time of the Baseline grows linearly with the number of updates, and due to its slow runtime, we only show the first 10 updates. However, the average query processing time for Leaf and Tree-OPT is less than 1s for both point and range query workloads, even for 200 updates. The results show that processing queries with the Leaf and Tree-OPT protocols provides a significant improvement over Baseline.

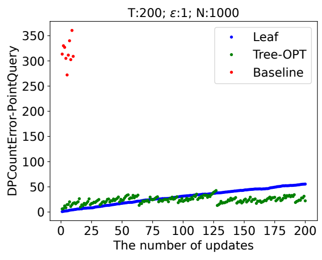


(a) Point query **(b) Range query**

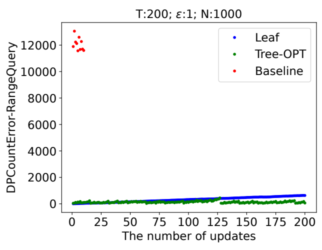
Figure 5: Run time of processing query

Observation 2. Leaf **and** Tree-OPT **have lower errors than** Baseline**.** Tree-OPT **shows lower errors than** Leaf **for a large number of updates.** Figure 6 shows error comparisons between Baseline, Leaf, and Tree-OPT. Baseline has significantly higher errors than Leaf and Tree-OPT for point and range queries workloads due to smaller privacy budgets at each update. Range queries have larger errors than point queries because they require the summation of noisy values over multiple bins. Both point and range queries have the same pattern of errors over time.

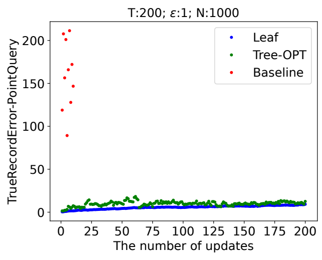
Figure 7 compares errors between Leaf and Tree-OPT. The largest number of updates is $T = 600$. Since the runtime for the Leaf protocol grows linearly with the number of updates (c.f. Figure 9a), we only run experiments for the first 200 updates. Leaf has an error linear in the number of updates (as also shown with the extrapolated line). On the other hand, Tree-OPT has an error logarithmic in the number of updates. When the number of updates is about 200, the error of Leaf begins exceed the error of Tree-OPT.



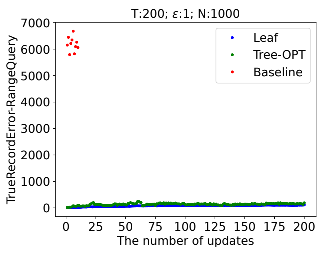
(a) Point query



(b) Range query



(c) Point query



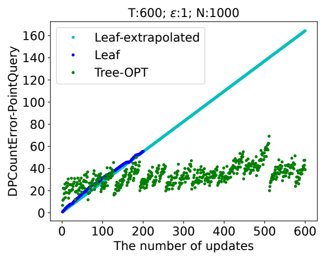
(d) Range query

Figure 6: Query errors of Leaf, Tree-OPT **and** Baseline

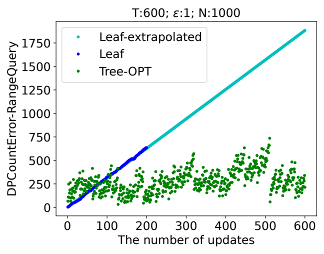
6.4 Performance Gain from Optimization

To evaluate Q2, we compare the speed-up of UpdateStores$_{\text{OPT}}$ versus UpdateStores$_{\text{ALL}}$ for updating the data stores. The results of Leaf, Tree-ALL, and Tree-OPT are shown as blue, red, and green points respectively in Figure 9a and 8.

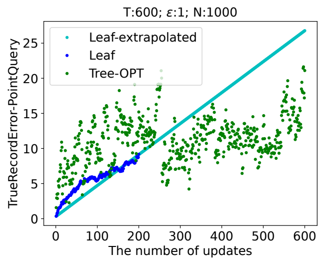
Observation 3. UpdateStores$_{\text{OPT}}$ **has a significantly shorter run time than** UpdateStores$_{\text{ALL}}$**, with slightly bigger errors.** We observe from Figure 9a that the running time of UpdateStores$_{\text{OPT}}$ is shorter than that of UpdateStores$_{\text{ALL}}$. When the number of updates is the n-th power of 2, UpdateStores$_{\text{OPT}}$ is significantly faster. This is because only part of the records in the previous data stores, which are highly likely to be dummy records, are sorted obliviously, rather than all records in these data stores. Thus, UpdateStores$_{\text{OPT}}$ dramatically reduces the number of records to be processed using MPC. Leaf does not sort records in previous data stores, but



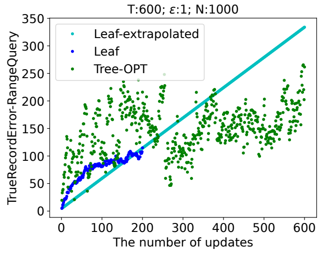
(a) Point query



(b) Range query



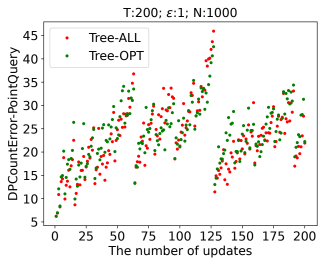
(c) Point query



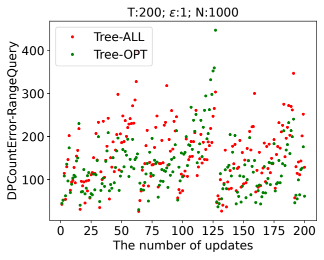
(d) Range query

Figure 7: Query errors of Leaf **and** Tree-OPT

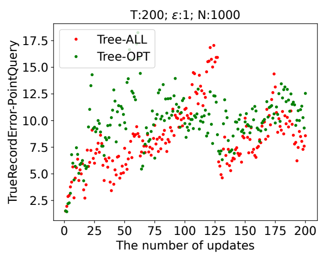
the number of dummy records accumulated in the deferred buffer grows linearly. Thus, the runtime of Leaf grows linearly. Note that Baseline takes negligible time to update data, since it simply appends new uploaded data to the data store. As shown in Figure 8, despite the optimizations leading to slightly higher errors, the error of Tree-OPT is comparable to Tree-ALL. The slightly higher TrueRecordError of Tree-OPT comes from not sorting all records in the previous data stores using UpdateStores$_{\text{OPT}}$, leading to dummy records accumulating in the updated store with low probability.



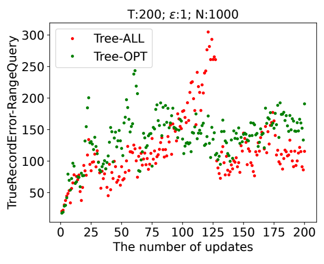
(a) Point query



(b) Range query

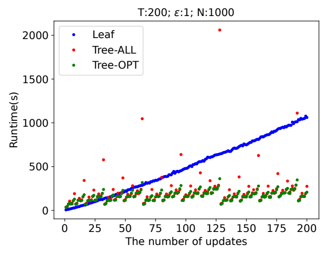


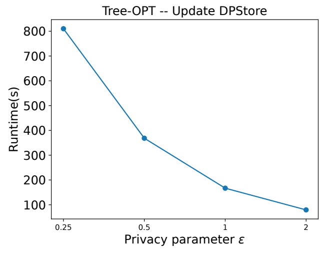
(c) Point query



(d) Range query

Figure 8: Query errors



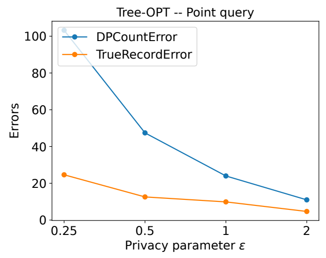


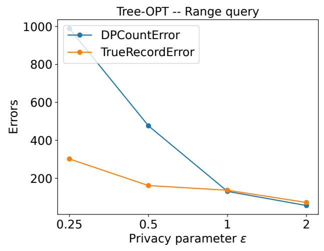
(a) Different approaches **(b) Different ϵ**

Figure 9: Run time of updating data store

6.5 Performance and Errors as a Function of ϵ

Observation 4. We address Q3 by evaluating Tree-OPT **with different ϵ in [0.25, 0.5, 1, 2].** Tree-OPT **exhibits different trade-offs of efficiency and accuracy for different privacy budgets.** As ϵ increases, we observe a half-decreasing trend in the query errors and the performance of Tree-OPT (c.f. Figures 9b and 10). Increasing the privacy budget results in less noise in Synopsis and fewer dummy records in ⟨Store⟩, leading to smaller query errors and faster running time.





(a) Point query **(b) Range query**

Figure 10: Query errors of Tree-OPT **with different ϵ**

7 EXTENSIONS

Multiple attributes. We now explain how to support queries with conditions on multiple queryable attributes (e.g. attributes A, B, C, D). There is extensive literature about releasing high-dimensional differentially private statistics [72]. Longshot can follow the marginal-based approach to securely materialize counts for a set of low-dimensional histograms (e.g., histogram over AB and histogram over CD) and post-processing them to measure a full-dimensional histogram (over all attributes A, B, C & D) to answer aggregate queries on the queryable attributes [56, 57, 79]. To return records for non-aggregate queries, there are two options. First, Longshot can sort records according to the ordered domain values of the full-dimensional histogram with all queryable attributes (e.g. A, B, C & D) and use that histogram to index records. The large number of bins required for the full-dimensional histogram incurs much larger storage maintenance costs. Second, instead of laying out records using the full-dimensional histogram, Longshot can maintain a copy of data and data structures for each low-dimensional histogram such that the records in each copy are sorted using the corresponding histogram (e.g. one for AB and one for CD). When processing a query, Longshot can satisfy a subset of conditions by using one of the low-dimensional histograms to fetch the appropriate records. For instance, a query on attributes A, B, & C would use the AB index. Longshot can then securely filter the fetched records to satisfy the conditions over the remaining attributes. This approach requires one copy of data for each low-dimensional histogram, incurring larger storage costs. Both approaches guarantee differentially private bounded leakage.

Multiple tables. Inspired by the idea in PrivateSQL [46], we can assume a representative workload of queries on multiple tables and generate a set of views to support them. Then we can use Longshot to generate data structures for each view, so that Longshot can answer the representation workload by processing queries using the data structures of a single view. We can bound the view sensitivity by dropping records that cause high sensitivity as in PrivateSQL or limiting the number of row that a record can contribute in a view as in incShrink [76]. However, how to design an update policy with bounded privacy loss to decide when to update all views together is a future research question to explore.

8 RELATED WORK

Secure databases. Many existing approaches incorporate cryptographically secure protocols into database management systems to allow executing queries on untrusted servers while maintaining strong security guarantees. These systems have used predicate encryption [52, 68], property and order preserving encryption [4, 10], symmetric searchable encryption (SSE) [19, 25, 41, 70], functional encryption [15, 67], oblivious RAM [23, 27, 59], multi-party secure computation (MPC) [7, 14, 71, 76], trusted execution environments (TEE) [31, 62, 73, 78] and homomorphic encryption [16, 21, 32, 65]. The security guarantees rely on hiding not just the value of the underlying records, but different leakage sources such as query patterns [77, 80], access patterns [26, 42], query response volume [35–37, 42], and leakage over time [5, 19, 33, 43, 75, 76]. Our approach uses one leakage function to hide all these leakages and design differentially-private data structures that significantly improve the performance of query execution.

Differential privacy for databases. A substantial body of research in differential privacy provides strong privacy guarantees for databases by protecting query results [3, 21, 24, 48, 53], or by protecting query execution [7, 13, 20, 54, 61, 66, 74]. Longshot utilizes differential privacy not only to bound information leakage but to improve the performance of cryptographic protocols over time by introducing novel oblivious algorithms that generate and maintain DP structures for multiple users on untrusted servers.

9 CONCLUSION

We introduce Longshot, the first privacy-preserving database management system that maintains DP data structures using MPC to support ad-hoc predicate queries on sensitive growing data continually uploaded by multiple, mutually distrustful clients. With Longshot, we introduce novel oblivious algorithms that balance the trade-off between privacy, and efficiency in a dynamic setting.

ACKNOWLEDGMENTS

This work was supported by NSF SaTC Award 2016393.

REFERENCES

[1] 2022. Emp-toolkit. https://github.com/emp-toolkit.

[2] 2022. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[3] Archita Agarwal, Maurice Herlihy, Seny Kamara, and Tarik Moataz. 2019. Encrypted Databases for Differential Privacy. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 170–190.

[4] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 563–574.

[5] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2019. Forward and backward private searchable encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.

[6] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: secure querying for federated databases. *arXiv preprint arXiv:1606.06808* (2016).

[7] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018), 307–320.

[8] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[9] Amos Beimel. 2011. Secret-sharing schemes: a survey. In *International conference on coding and cryptology*. Springer, 11–46.

[10] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*. Springer, 535–552.

[11] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2017. The tao of inference in privacy-protected databases. *Cryptology ePrint Archive* (2017).

[12] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2019. Revisiting Leakage Abuse Attacks. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1175.

[13] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. ϵpsolute : Efficiently Querying Databases While Providing Differential Privacy. *arXiv preprint arXiv:1706.01552* (2021).

[14] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1175–1191.

[15] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. 2004. Public key encryption with keyword search. In *International conference on the theory and applications of cryptographic techniques*. Springer, 506–522.

[16] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. 2005. Evaluating 2-DNF formulas on ciphertexts. In *Theory of cryptography conference*. Springer, 325–341.

[17] Yang Cao, Masatoshi Yoshikawa, Yonghui Xiao, and Li Xiong. 2018. Quantifying differential privacy in continuous data release under temporal correlations. *IEEE transactions on knowledge and data engineering* 31, 7 (2018), 1281–1295.

[18] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 668–679.

[19] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: data structures and implementation.. In *NDSS*, Vol. 14. Citeseer, 23–26.

[20] Guoxing Chen, Ten-Hwang Lai, Michael K Reiter, and Yinqian Zhang. 2018. Differentially private access patterns for searchable symmetric encryption. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 810–818.

[21] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2019. Crypt$epsilon$: Crypto-Assisted Differential Privacy on Untrusted Servers. *arXiv preprint arXiv:1902.07756* (2019).

[22] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics.. In *NSDI*. 259–282.

[23] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 727–743. https://www.usenix.org/conference/osdi18/presentation/crooks

[24] Rachel Cummings, Sara Krehbiel, Kevin A Lai, and Uthaipon Tantipongpipat. 2018. Differential privacy for growing databases. *arXiv preprint arXiv:1803.06416* (2018).

[25] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security* 19, 5 (2011), 895–934.

[26] Jonathan L Dautrich Jr and Chinya V Ravishankar. 2013. Compromising privacy in precise query protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*. 155–166.

[27] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. {SEAL}: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[28] Irit Dinur and Kobbi Nissim. 2003. Revealing information while preserving privacy. *PODS*.

[29] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[30] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N Rothblum. 2010. Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*. 715–724.

[31] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing using hardware enclaves. *arXiv preprint arXiv:1710.00458* (2017).

[32] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[33] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1038–1055.

[34] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[35] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 315–331.

[36] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1067–1083.

[37] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 361–378.

[38] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. 2009. Boosting the accuracy of differentially-private histograms through consistency. *arXiv preprint arXiv:0904.0942* (2009).

[39] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *Ndss*, Vol. 20. Citeseer, 12.

[40] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2014. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. 235–246.

[41] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 965–976.

[42] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.

[43] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2017. Accessing data while preserving privacy. *arXiv preprint arXiv:1706.01552* (2017).

[44] Georgios Kellaris, Stavros Papadopoulos, Xiaokui Xiao, and Dimitris Papadias. 2014. Differentially private event sequences over infinite streams. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1155–1166.

[45] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1223–1240.

[46] 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.

[47] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.

[48] Mathias Lécuyer, Riley Spahn, Kiran Vodrahalli, Roxana Geambasu, and Daniel Hsu. 2019. Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 181–195. https://doi.org/10.1145/3341301.3359639

[49] Jaewoo Lee, Yue Wang, and Daniel Kifer. 2015. Maximum likelihood postprocessing for differential privacy under consistency constraints. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 635–644.

[50] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. 2015. The matrix mechanism: optimizing linear counting queries under differential privacy. *VLDB*.

[51] Haoran Li, Li Xiong, Xiaoqian Jiang, and Jinfei Liu. 2015. Differentially private histogram publication for dynamic datasets: an adaptive sampling approach.

In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 1001–1010.

[52] Yanbin Lu. 2012. Privacy-preserving Logarithmic-time Search on Encrypted Data in Cloud.. In *NDSS*.

[53] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. 2021. Privacy Budget Scheduling. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 55–74.

[54] Sahar Mazloom and S Dov Gordon. 2018. Secure computation with differentially private access patterns. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 490–507.

[55] Ryan McKenna, Gerome Miklau, Michael Hay, and Ashwin Machanavajjhala. 2018. Optimizing error of high-dimensional statistical queries under differential privacy. *VLDB*.

[56] Ryan McKenna, Gerome Miklau, and Daniel Sheldon. 2021. Winning the NIST Contest: A scalable and general approach to differentially private synthetic data. *arXiv preprint arXiv:2108.04978* (2021).

[57] Ryan McKenna, Daniel Sheldon, and Gerome Miklau. 2019. Graphical-model based estimation and inference for differential privacy. In *International Conference on Machine Learning*. PMLR, 4435–4444.

[58] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 644–655.

[59] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. 2014. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 639–654.

[60] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 377–394.

[61] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 79–93.

[62] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[63] Wahbeh Qardaji, Weining Yang, and Ninghui Li. 2013. Understanding hierarchical methods for differentially private histograms. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1954–1965.

[64] Paul Grubbs Tom Ristenpart and Vitaly Shmatikov. [n.d.]. Why Your Encrypted Database Is Not Secure. ([n. d.]).

[65] Bharath Kumar Samanthula, Wei Jiang, and Elisa Bertino. 2014. Privacy-preserving complex query evaluation over semantically secure encrypted data. In *European Symposium on Research in Computer Security*. Springer, 400–418.

[66] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. 2021. Obfuscated Access and Search Patterns in Searchable Encryption. *arXiv preprint arXiv:2102.09651* (2021).

[67] Emily Shen, Elaine Shi, and Brent Waters. 2009. Predicate privacy in encryption systems. In *Theory of Cryptography Conference*. Springer, 457–473.

[68] Elaine Shi, John Bethencourt, TH Hubert Chan, Dawn Song, and Adrian Perrig. 2007. Multi-dimensional range query over encrypted data. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 350–364.

[69] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[70] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 71. 72–75.

[71] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CRYPTGPU: Fast Privacy-Preserving Machine Learning on the GPU. *arXiv preprint arXiv:2104.10949* (2021).

[72] Yuchao Tao, Ryan McKenna, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2021. Benchmarking differentially private synthetic data generation algorithms. *arXiv preprint arXiv:2112.09238* (2021).

[73] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. Stealthdb: a scalable encrypted database with full SQL query support. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 370–388.

[74] Sameer Wagh, Paul Cuff, and Prateek Mittal. 2018. Differentially private oblivious ram. *Proceedings on Privacy Enhancing Technologies* 2018, 4 (2018), 64–84.

[75] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. DP-Sync: Hiding Update Patterns in Secure OutsourcedDatabases with Differential Privacy. *arXiv preprint arXiv:2103.15942* (2021).

[76] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: architecting efficient outsourced databases using incremental mpc and differential privacy. *SIGMOD*.

[77] Xingchen Wang and Yunlei Zhao. 2018. Order-revealing encryption: file-injection attack and forward security. In *European Symposium on Research in Computer Security*. Springer, 101–121.

[78] Min Xu, Antonis Papadimitriou, Andreas Haeberlen, and Ariel Feldman. 2019. Hermetic: Privacy-preserving distributed analytics without (most) side channels. *External Links: Link Cited by* (2019).

[79] Jun Zhang, Graham Cormode, Cecilia M Procopiuc, Divesh Srivastava, and Xiaokui Xiao. 2017. Privbayes: Private data release via bayesian networks. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–41.

[80] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 707–720.