IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy

Chenghong Wang
Duke University
chwang@cs.duke.edu

Johes Bater
Duke University
johes.bater@duke.edu

Kartik Nayak
Duke University
kartik@cs.duke.edu

Ashwin Machanavajjhala
Duke University
ashwin@cs.duke.edu

ABSTRACT

In this paper, we consider secure outsourced growing databases (SOGDB) that support view-based query answering. These databases allow untrusted servers to privately maintain a materialized view. This allows servers to use only the materialized view for query processing instead of accessing the original data from which the view was derived. To tackle this, we devise a novel view-based SOGDB framework, IncShrink. The key features of this solution are: (i) IncShrink maintains the view using incremental MPC operators which eliminates the need for a trusted third party upfront, and (ii) to ensure high performance, IncShrink guarantees that the leakage satisfies DP in the presence of updates. To the best of our knowledge, there are no existing systems that have these properties. We demonstrate IncShrink's practical feasibility in terms of efficiency and accuracy with extensive experiments on real-world datasets and the TPC-ds benchmark. The evaluation results show that IncShrink provides a 3-way trade-off in terms of privacy, accuracy and efficiency, and offers at least a 7,800× performance advantage over standard SOGDB that do not support view-based query paradigm.

CCS CONCEPTS

• **Security and privacy → Data anonymization and sanitization**; **Management and querying of encrypted data**.

KEYWORDS

Incremental MPC; Differential privacy; Secure outsourced database

ACM Reference Format:
Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526151

1 INTRODUCTION

There is a rapid trend of organizations moving towards outsourcing their data to cloud providers to take advantages of its cost-effectiveness, high availability, and ease of maintenance. Secure outsourced databases are designed to help organizations outsource their data to untrusted cloud servers while providing secure query functionalities without sacrificing data confidentiality and privacy. The main idea is to have the data owners upload the encrypted or secret-shared data to the outsourcing servers. Moreover, the servers are empowered with secure protocols which allow them to process queries over such securely provisioned data. A series of works such as CryptDB [69], Cipherbase [4], EnclaveDB [70], and HardIDX [33] took the first step in the exploration of this scope by leveraging strong cryptographic primitives or secure hardware to accomplish the aforementioned design goals. Unfortunately, these solutions fail to provide strong security guarantees, as recent works on leakage-abuse attacks [10, 19, 49, 88] have found that they are vulnerable to a variety of reconstruction attacks that exploit side-channel leakages. For instance, an adversary can fully reconstruct the data distribution after observing the query processing transcripts.

Although some recent efforts, such as [7, 11, 28, 32, 46, 65, 67, 68, 87, 89], have shown potential countermeasures against leakage-abuse attacks, the majority of these works focus primarily on static databases. A more practical system often requires the support of updates to the outsourced data [1, 35, 78, 83], which opens up new challenges. Wang et al. [83] formulate a new type of leakage called *update pattern* that affects many existing outsourced database designs when the underlying data is dynamically growing. To mitigate such weakness, their solution dictates the data owners' update behavior to private record synchronization strategies, with which it perturbs the owners' logical update pattern. However, their solution only considers a naïve query answering mode such that each query is processed independently and evaluated directly over the entire outsourced data. This inevitably leads to a substantial amount of redundant computation by the servers. For example, consider the following use case where a courier company partners with a local retail store to help deliver products. The retail store has its sales data, and the courier company has its delivery records, both of which are considered to be the private property of each. Now assume the retail store owner wants to know how many of her products are delivered on time (i.e., within 48 hours of the courier accepting the package). With secure outsourced databases, the store owner and the courier company have the option to securely outsource their data and its corresponding computations to cloud servers. However, in a naïve query processing mode, the servers have to recompute the entire join relation between the outsourced data whenever a query is posted, which raises performance concerns.

In this work, we take the next step towards designing a secure outsourced growing database (SOGDB) architecture with a more efficient query answering mechanism. Our proposed framework employs a novel secure query processing method in which the

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526151

servers maintain a growing size materialized view corresponding to the owner's outsourced data. The upcoming queries will be properly answered using only the materialized view object. This brings in inherent advantages of view-based query answering [77] paradigm, such as allowing the servers to cache important intermediate outputs, thus preventing duplicated computation. For instance, with our view-based SOGDB architecture, one can address the performance issues in the aforementioned use case by requiring the servers to maintain a materialized join table between the outsourced sales and delivery data. Moreover, whenever the underlying data changes, the materialized join table is updated accordingly. To this end, the servers only need to perform secure filtering over the materialized join table for processing queries, which avoids duplicated computation of join relations.

There is no doubt one can benefit in many aspects from the view-based query answering paradigm. However, designing a practical view-based SOGDB is fraught with challenges. First, the servers that maintain the materialized view is considered to be potentially untrusted. Hence, we must explore the possibility of updating such materialized view without a trusted curator. A typical way is to leverage secure multi-party computation (MPC). However, naïvely applying MPC for updating view instances over growing data would expose extra information leakage (i.e., update pattern [83]). For example, consider the use case we mentioned before where the servers maintain a join table over the sales and the delivery data. Even with MPC, one can still obtain the true cardinality of newly inserted entries to the join table by looking at the output size from MPC. This would allow the adversary to learn the exact number of packages requested for delivery by the local retail store at any given time. Naïve methods, such as always padding newly generated view tuples to the maximum possible size or choosing never to update the materialized view, could prevent the aforementioned leakage. However, such an approach either introduces a large performance burden or does not provide us with the functionality of database updates. To combat this, we propose a novel view update methodology that leverages incremental MPC and differential privacy (DP), which hides the corresponding update leakage using DP while balancing between the efficiency and accuracy.

This design pattern helps us to address the extra leakage, but also raises new challenges. The transformation from outsourced data to a view instance may have unbounded stability, i.e., an input record may contribute to the generation of multiple rows in the transformed output, which could cause unbounded privacy loss. To address this, we enforce that any individual data outsourced by the owner only contributes to the generation of a fixed number of view tuples. As the transformation after applying this constraint has bounded stability, thus we obtain a fixed privacy loss with respect to each insertion (logical update) to the owner's logical data.

Putting all these building blocks together, a novel view-based SOGDB framework, IncShrink, falls into place. We summarize our contributions as follows:

- IncShrink is a first of its kind, secure outsourced growing database framework that supports view-based query processing paradigm. Comparing with the standard SOGDB [83] that employs naïve query answering setting, IncShrink improves query efficiency, striking a balance between the guarantees of privacy, efficiency and accuracy, at the same time.
- IncShrink integrates incremental MPC and DP to construct the view update functionality which (i) allows untrusted entities to securely build and maintain the materialized view instance (ii) helps to reduce the performance overhead of view maintenance, and (iii) provides a rigorous DP guarantee on the leakage revealed to the untrusted servers.
- IncShrink imposes constraints on the record contribution to view tuples which ensures the entire transformation from outsourced data to the view object over time has bounded stability. This further implies a bounded privacy loss with respect to each individual logical update.
- We evaluate IncShrink on use cases inspired by the *Chicago Police Data* and the TPC-ds benchmark. The evaluation results show at least 7800× and up to 1.5e+5× query efficiency improvement over standard SOGDB. Moreover, our evaluation shows that IncShrink provides a 3-way trade-off between privacy, efficiency, and utility while allowing users to adjust the configuration to obtain their desired guarantees.

2 OVERVIEW

We design IncShrink to meet three main goals:

- **View-based query answering.** IncShrink enables view-based query answering for a class of specified queries over secure outsourced growing data.
- **Privacy against untrusted server.** Our framework allows untrusted servers to continuously update the materialized view while ensuring that the privacy of the owners' data is preserved against outsourcing servers.
- **Bounded privacy loss.** The framework guarantees an unlimited number of updates under a fixed privacy loss.

In this section, we first outline the key ideas that allow IncShrink to support the our research goals (Section 2.1). Then briefly review the framework components in Section 2.2. We provide a running example in Section 2.3 to illustrate the overall framework workflow.

2.1 Key Ideas

KI-1. View-based query processing over secure outsourced growing data. IncShrink employs materialized view for answering pre-specified queries over secure outsourced growing data. The framework allows untrusted outsourcing servers to securely build and maintain a growing-size materialized view corresponding to the selected view definition. A typical materialized view can be either transformed solely based on the data provisioned by the owners, i.e., a join table over the outsourced data, or in combination with public information, i.e., a join table between the outsourced data and public relations. Queries posed to the servers are rewritten as queries over the defined view and answered using only the view object. Due to the existence of materialization, the outsourcing servers are exempted from performing redundant computations.

KI-2. Incremental MPC with DP update leakage. A key design goal of IncShrink is to allow the untrusted servers to privately update the view instance while also ensuring the privacy of owners' data. As mentioned before, compiling the view update functionality into the MPC protocol is not sufficient to ensure data privacy, as it still leaks the true cardinality of newly inserted view entries at

each time, which is directly tied with the owners' record update patterns [83]. Although naïve approaches such as exhaustive padding (EP) of MPC outputs or maintaining a one-time materialized view (OTM) could alleviate the aforementioned privacy risk. They are known to either incorporate a large amount of dummy data or provide poor query accuracy due to the lack of updates to the materialized view. This motivates our second key idea to design an incremental MPC protocol for view updates while balancing the privacy, accuracy, and efficiency guarantees.

In our design, we adopt an innovative "Transform-and-Shrink" paradigm, where the protocol is composed of two sub-protocols, Transform and Shrink, that coordinate with each other. Transform generates corresponding view entries based on newly outsourced data, and places them to an exhaustively padded secure cache to avoid information leakage. Shrink, runs independently, periodically synchronizes the cached data to the materialized view. To prevent the inclusion of a large amount of dummy data, Shrink shrinks the cached data to a DP-sized secure array such that a subset of the dummy data is removed whereas the true cardinality is still preserved. As a result, the resulting protocol ensures any entity's knowledge with respect to the view instance is bounded by DP.

KI-3. Fixed privacy loss through constraints on record contributions. When IncShrink releases noisy cardinalities, it ensures ϵ-DP with respect to the view instance. However, this does not imply ϵ-DP to the logical data where the view is derived. This is because an individual data point in the logical database may contribute to generating multiple view entries. As a result, the framework either incurs an unbounded privacy loss or has to stop updating the materialized view after sufficiently many synchronizations. This leads us to our third key idea, where we bound the privacy loss by imposing constraints on the contributions made by each individual record to the generation of the view object. Each data point in the logical database is allocated with a contribution budget, which is consumed whenever the data is used to generate a new view entry. Once the contribution budget for a certain record is exhausted, IncShrink retires this data and will no longer use it to generate view entries. With such techniques, IncShrink is able to constantly update the materialized view with a bounded privacy loss. On the other hand, despite such constraints, IncShrink is still able to support a rich class of queries with small errors (Section 7).

2.2 Framework Components

Underlying database. IncShrink does not create a new secure outsourced database but rather builds on top of it. Therefore, as one of the major components, we assume the existence of an underlying secure outsourced database scheme. Typically, secure outsourced databases can be implemented according to different architectural settings, such as the models utilizing server-aided MPC [6, 7, 47, 64, 79], homomorphic encryption [22], symmetric searchable encryption [3, 9, 20, 26, 35, 48, 78] or trusted hardware [32, 70, 81, 89]. For the ease of demonstration, we focus exclusively on the outsourced databases built upon the server-aided MPC model, where a set of data owners secretly share their data to two untrusted but non-colluding servers $\mathcal{S}_0$ and $\mathcal{S}_1$. The two outsourcing servers are able to perform computations (i.e., query processing) over the secret shared data by jointly evaluating a 2-party secure computation protocol. More details about this setting and its corresponding security definitions are provided in Section 4. We stress that, although the protocols described in this paper assumes an underlying database architected under the server-aided MPC setup, these protocols can be adapted to other settings as well.

Materialized view. A materialized view is a subset of a secure outsourced database, which is typically generated from a query and stored as an independent object (i.e., an encrypted or secret-shared data structure). The servers can process queries over the view instance just as they would in a persistent secure database. Additionally, changes to the underlying data are reflected in the entries shown in subsequent invocations of the materialized view.

View update protocol. The view update protocol is an incremental MPC protocol jointly evaluated by the outsourcing servers. It allows untrusted servers to privately update the materialized view with bounded leakage. More details can be found in Section 5

Secure outsourced cache. The secure outsourced cache is a secure array (i.e., memory blocks that are encrypted, secret-shared, or stored inside trusted hardware) denoted as $\sigma[1, 2, 3, \ldots]$, which is used to temporarily store newly added view entries that will later be synchronized to the materialized view. In this work, as we focus on the server-aided MPC model, thus σ is considered as a secret shared memory block across two non-colluding servers. Each $\sigma[i]$ represents a (secret-shared) view entry or a dummy tuple. Details on how our view update protocol interacts with the secure cache (i.e., read, write, and flush cache) are provided in Section 5.

2.3 IncShrink Workflow

We now briefly review the framework architecture and its workflow with a running example (Figure 1), where an analyst is interested in a join query over the outsourced data from two data owners.

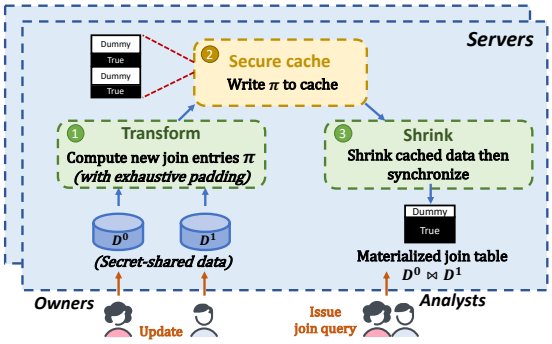


Figure 1: Framework workflow.

Initially, the analyst obtains authentications from the owners and registers the query with the outsourcing servers. The servers decide the view definition as a join table, set up the initial materialization structure, the secure cache, and compile the corresponding secure protocols Transform and Shrink for maintaining the view instance. Since then, the owners periodically update the outsourced data, securely provisioning the newly received data since the last update (through the functionality defined by the underlying database). For demonstration purposes, we assume the owners submit a fixed-size data block (possibly padded with dummy records) at predetermined intervals. We discuss potential extensions to support other update behaviors in a later section. Whenever owners submit new data, the servers invoke Transform to securely compute new joins. The

outputs will be padded to the maximum size then placed in a secure cache. Next, Shrink periodically synchronizes data from the cache to the join table with DP resized cardinalities. Note that the DP noise used to distort the true cardinality can be either positive or negative. If the noise is negative, some of the real tuples in the cache are not fetched by Shrink. We refer these tuples as the "deferred data". On the contrary, if the noise is positive, some deferred data or additional dummy tuples will be included to synchronize with the view. On the other hand, the analyst can issue query requests to the servers, which process the issued queries over the materialized join table and return the resulting outputs back to the analyst.

3 PRELIMINARIES

Multi-party secure computation (MPC). MPC utilizes cryptographic primitives to enable a set of participants $P_1, P_2, ..., P_n$ to jointly compute a function f over private input data x_i supplied by each party P_i, without using a trusted third party. The theory of MPC offers strong security guarantee similar as what can be achieved with a trusted third party [36], i.e., absolutely no information leak to each participant P_i beyond the desired output of $f(x_1, x_2, ..., x_n)$ and their input x_i. In this work we focus mainly on the 2-party secure computing setting.

(n, t)-secret sharing. Given ring $\mathbb{Z}_m$, and $m = 2^\ell$. A (n, t)-secret sharing (t-out-of-n) over $\mathbb{Z}_m$ shares a secret value $x \in \mathbb{Z}_m$ with n parties such that the sharing satisfies the following property:

- **Availability** Any t' of the n parties such that $t' \geq t$ can recover the secret value x.
- **Confidentiality** Any t' of the n parties such that $t' < t$ have no information of x.

For any value $x \in \mathbb{Z}_m$, we denote it's secret sharing as $[\![x]\!]^m \leftarrow (x_1, x_2, ..., x_n)$. There are many existing efforts to implement such secret sharing design [8], we focus on XOR-based $(2, 2)$-secret sharing over $\mathbb{Z}_{2^{32}}$ with the following specifications.

- **Generate shares** share(x): Given $x \in \mathbb{Z}_m$, sample random values $x_1 \xleftarrow{\text{rd}} \mathbb{Z}_m$, compute $x_2 \leftarrow x \oplus x_1$, and return secret shares $[\![x]\!]^m \leftarrow (x_1, x_2)$.
- **Recover shares** recover$([\![x]\!]^m)$: Given secret shares $[\![x]\!]^m \leftarrow (x_1, x_2)$, compute $x \leftarrow x_1 \oplus x_2$, then return x.

4 PRIVACY MODEL

In general, we consider our framework supports dynamic updating of the materialized view while hiding the corresponding update leakage. More specifically, we consider the participants involved in the outsourcing phase are a set of data owners and two servers $\mathcal{S}_0$, and $\mathcal{S}_1$. We assume there exists a semi-honest *probabilistic polynomial time (p.p.t.)* adversary $\mathcal{A}$ who can corrupt any subset of the owners and at most one of the two servers. Previous work [64] refers to this type of adversary as the *admissible adversary*, which captures the property of two non-colluding servers, i.e., if one is compromised by the adversary, the other one behaves honestly. Our privacy definition requires that the knowledge $\mathcal{A}$ can obtain about any single data of the remaining honest owners, by observing the view updates, is bounded by differential privacy. In this section, we first provide key terminologies and notations (Section 4.1) then formalize our privacy model (Section 4.2) using *simulation-based computational differential privacy (SIM-CDP)* [63].

4.1 Notations

Growing database. A growing database is a dynamic relational dataset with insertion only updates, thus we define it as a collection of (logical) updates, $\mathcal{D} = \{u_i\}_{i \geq 0}$, where u_i is a time stamped data. We write $\mathcal{D} = \{\mathcal{D}_t\}_{t \geq 0}$, such that $\mathcal{D}_t$ denotes the database instance of the growing database $\mathcal{D}$ at time t and $\forall\ \mathcal{D}_t\ , \mathcal{D}_t \subseteq \mathcal{D}$.

Outsourced data. The outsourced data is denoted as $\mathcal{DS}$, which stores the secret-shared entries corresponding to the records in the logical database, with the possibility to include additional dummy data. Similarly, we write $\mathcal{DS} = \{\mathcal{DS}_t\}_{t \geq 0}$, where $\mathcal{DS}_t \subseteq \mathcal{DS}$ is the outsourced data at time t.

Materialized view. We use $\mathcal{V}$ to denote the materialized view instance which is a collection of secret-shared tuples. Each tuple in $\mathcal{V}$ is transformed from the outsourced data $\mathcal{DS}$ or in combination with public information. We define $\mathcal{V} = \{\mathcal{V}_t\}_{t \geq 0}$, where $\mathcal{V}_t$ denotes the materialized view structure at time t, and $\Delta\mathcal{V}_t$ denotes the changes between (newly generated view entries) $\mathcal{V}_t$ and $\mathcal{V}_{t-1}$

Query. Given a growing database $\mathcal{D}$ and a corresponding materialized view $\mathcal{V}$, we define the logical query posted at time t as $q_t(\mathcal{D}_t)$ and the re-written view-based query as $\tilde{q}_t(\mathcal{V}_t)$. We refer the L1 norm of the difference between $\tilde{q}_t(\mathcal{V}_t)$ and $q_t(\mathcal{D}_t)$ as the *L1 query error*, denoted as $L_{q_t} \leftarrow ||\tilde{q}_t(\mathcal{V}_t) - q_t(\mathcal{D}_t)||_1$, which measures the difference between the server responded outputs and their corresponding logical results. Additionally, we call the elapsed time for processing $\tilde{q}_t(\mathcal{V}_t)$ as the *query execution time (QET)* of q_t. In this work, we use L1 error and QET as the main metrics to evaluate the accuracy and efficiency of our framework, respectively.

4.2 Privacy Definition

Based on the formalization of update pattern in [83], we first provide a more generalized definition of update pattern that captures updates to view instances.

Definition 1 (Update pattern). *Given a growing database $\mathcal{D}$, the update pattern for outsourcing $\mathcal{D}$ is the function family of* $\mathsf{UpdtPatt}(\mathcal{D}) = \{\mathsf{UpdtPatt}_t(\mathcal{D})\}_{t \in \mathbb{N}^+}$, *with:*

$$\mathsf{UpdtPatt}_t(\mathcal{D}) = (t, |T_t(\mathcal{D})|)$$

where T_t is a transformation function that outputs a set of tuples (i.e., new view entries) been outsourced to the server at time t.

In general, Definition 1 defines the transcript of entire update history for outsourcing a growing database $\mathcal{D}$. It may include information about the volume of the outsourced data and their corresponding insertion times. Moreover, if $T_t(\mathcal{D}) \leftarrow \mathcal{D}_t - \mathcal{D}_{t-1}$, then this simply indicates the record insertion pattern [83].

Definition 2 (Neighboring growing databases). *Given a pair of growing databases $\mathcal{D}$ and $\mathcal{D}'$, such that there exists some parameter $\tau \geq 0$, the following holds: (i) $\forall\ t \leq \tau, \mathcal{D}_t = \mathcal{D}'_t$ (ii) $\forall\ t > \tau$, $\mathcal{D}_t$ and $\mathcal{D}'_t$ differ by the addition or removal of a single logical update.*

Definition 3 (DP mechanism over growing data). *Let F to be a mechanism applied over a growing database. F is said to satisfy*

ϵ-DP if for any neighboring growing databases $\mathcal{D}$ and $\mathcal{D}'$, and any $O \in \mathcal{O}$, where $\mathcal{O}$ is the universe of all possible outputs, it satisfies:

$$\Pr\left[F(\mathcal{D}) \in O\right] \leq e^{\epsilon}\Pr\left[F(\mathcal{D}') \in O\right] \tag{1}$$

Definition 3 ensures that, by observing the output of F, the information revealed by any single logical update posted by the owner is differentially private. Moreover, if the logical update corresponds to different entity's (owner's) data, the same holds for F over each owner's logical database (privacy is guaranteed for each entity). Additionally, in this work, we assume each logical update $u_i \in \mathcal{D}$ as a secret event, therefore such mechanism F achieves ϵ-event level DP [30]. However, due to the group-privacy property [53, 58, 86] of DP, one can achieve privacy for properties across multiple updates as well as at the user level as long as the property depends on a finite number of updates. An overall ϵ-user level DP can be achieved by setting the privacy parameter in Definition 3 to $\frac{\epsilon}{\ell}$, where ℓ is the maximum number of tuples in the growing database owned by a single user. In practice, if the number of tuples owned by each user is unknown, a pessimistic large value can be chosen as k. Moreover, recent works [18, 76] have provided methods for deriving an $\epsilon < \epsilon' \leq \ell \times \epsilon$, such that ϵ-event DP algorithms provide an ϵ' bound on privacy loss when data are correlated. For certain correlations, ϵ' can be even close to ϵ and much smaller than $\ell \times \epsilon$. In general, we emphasize that for the remainder of the paper, we focus exclusively on developing algorithms that ensure event-level privacy with parameter ϵ, while simultaneously satisfying all the aforementioned privacy guarantees, with possibly a different privacy parameter.

Definition 4 (SIM-CDP view update protocol). *A view update protocol Π is said to satisfy ϵ-SIM-CDP if there exists a p.p.t. simulator $\mathcal{S}$ with only access to a set of public parameters* pp *and the output of a mechanism F that satisfies Definition 3. Then for any growing database instance $\mathcal{D}$, and any p.p.t. adversary $\mathcal{A}$, the adversary's advantage satisfies:*

$$\begin{aligned}&\Pr\left[\mathcal{A}\left(\mathsf{VIEW}^{\Pi}(\mathcal{D}, \mathsf{pp}) = 1\right)\right] \\ &\leq \Pr\left[\mathcal{A}\left(\mathsf{VIEW}^{\mathcal{S}}(F(\mathcal{D}), \mathsf{pp})\right) = 1\right] + \mathsf{negl}(\kappa)\end{aligned} \tag{2}$$

where VIEW^{Π}, and $\mathsf{VIEW}^{\mathcal{S}}$ denotes the adversary's view against the protocol execution and the simulator's outputs, respectively. And $\mathsf{negl}(\kappa)$ is a negligible function related to a security parameter κ.

Definition 4 defines the secure protocol for maintaining the materialized view such that as long as there exists at least one honest owner, the privacy of her data's individual records is guaranteed. In addition, the remaining entities' knowledge about honest owner's data is preserved by differential privacy. In other words, any *p.p.t.* adversary's knowledge of such protocol Π is restricted to the outputs of an ϵ-DP mechanism F. We refer to the mechanism F as the leakage profile of protocol Π, and a function related to the update pattern, i.e., $F(\mathcal{D}) = f(\mathsf{UpdtPatt}(\mathcal{D}))$. In the rest of this paper, we focus mainly on developing view update protocols that satisfy this definition. Moreover, Definition 4 is derived from the SIM-CDP definition which is formulated under the *Universal Composition (UC)* framework [17]. Thus Definition 4 leads to a composability property such that if other protocols (i.e., Query protocol) defined by the underlying databases also satisfy UC security, then privacy guarantee holds under the composed system.

5 PROTOCOL DESIGN

In general, our view update protocol is implemented as an incremental MPC across two non-colluding outsourcing servers. Specifically, this incremental MPC is composed of two sub-protocols, Transform, and Shrink that operate independently but collaborate with each other. The reason for having this design pattern is that we can decouple the view transformation functionality and its update behavior, which provides flexibility in the choice of different view update strategies. For example, one may want to update the materialized view at a fixed interval or update it when there are enough new view entries. Each time when one needs to switch between these two strategies, she only needs to recompile the Shrink protocol without making any changes to the Transform protocol. In this section, we discuss the implementation details of these two protocols, in Section 5.1and 5.2, respectively. Due to space concerns, for theorems in this section, we defer the proofs to the appendix.

5.1 Transform Protocol

Whenever owners submit new data, Transform is invoked to convert the newly outsourced data to its corresponding view instance based on a predefined view definition. Although, one could simply reuse the query capability of the underlying database to generate the corresponding view tuples. There are certain challenges in order to achieve our targeted design objectives. Here are two examples: (i) The view transformation might have unbounded stability which further leads to an unbounded privacy loss; (ii) While existing work [7] implements a technique similar to first padding the output and then reducing its size, they compile the functionality as a one-time MPC protocol, which makes it difficult for them to handle dynamic data. Our design of constructing "Transform" and "Shrink" as independent MPC protocols overcome this problem and introduce flexibility in the choice of view update policy, but it raises a new challenge in that the two independently operating protocols still need to collaborate with each other. By default, the Shrink protocol is unaware of how much data can be removed from the secure cache, therefore Transform must privately inform Shrink how to eliminate the dummy records while ensuring DP. To address these challenges, we employ the following techniques:

- We adopt a truncated view transformation functionality to ensure that each outsourced data contributes to a bounded number of rows in the transformed view instance.
- We track important parameters in the Transform phase, secretly share them inside the Transform protocol and store the corresponding shares onto each server. The parameters are later passed to Shrink protocol as auxiliary input and used to generate the DP resized cardinalities.

Algorithm 1 provides an overview of Transform protocol. At the very outset, Transform is tasked to: (i) convert the newly submitted data into its corresponding view entry from time to time; (ii) cache the new view entries to an exhaustively padded secure array; and (iii) maintain a cardinality counter of how many new view entries have been cached since the last view update. This counter is then privately passed (through secret shares) to the Shrink protocol.

At the very beginning, Transform initializes the cardinality counter $c = 0$ and secret shares it to both servers (Alg 1:1-2). Transform uses trans_truncate (Alg 1:3) operator to obliviously compute the

Algorithm 1 Transform protocol

Input: ω (truncation bound); $\mathcal{DS}_t$; $\boldsymbol{\sigma}$.
1: **if** t == 0 **then**
2: $\quad [\![c]\!]^m = (x \xleftarrow{\text{rd}} \mathbb{Z}_m, x \oplus 0), [\![c]\!]^m \Rightarrow (\mathcal{S}_0, \mathcal{S}_1)$
3: $\Delta\mathcal{V} \leftarrow \text{trans\_truncate}(\mathcal{DS}_t, \omega)$
4: $[\![c]\!]^m \Leftarrow (\mathcal{S}_0, \mathcal{S}_1), c \leftarrow \text{recover}([\![c]\!]^m)$
5: $c \leftarrow c + \sum_{v_i \in \Delta\mathcal{V} \wedge v_i \neq \text{dummy}} \mathbb{1}$
6: $[\![c]\!]^m \leftarrow \text{share}(c), [\![c]\!]^m \Rightarrow (\mathcal{S}_0, \mathcal{S}_1)$
7: $\boldsymbol{\sigma} \leftarrow \boldsymbol{\sigma} || \Delta\mathcal{V}$

new view tuples and truncate the contribution of each record at the same time. More specifically, we assume the output of this operator is stored in an exhaustively padded secure array $\Delta\mathcal{V}$, where each $\Delta\mathcal{V}[i]$ is a transformed tuple with an extra *isView* bit to indicate whether the corresponding tuple is a view entry (*isView=1*) or a dummy tuple (*isView=0*). Additionally, the operator ensures that

$$\forall ds_i \in \mathcal{DS}_t, ||g^{\omega}(\mathcal{DS}_t) - g^{\omega}(\mathcal{DS}_t - \{ds_i\})|| \leq \omega \quad (3)$$

where $g^{\omega}(\cdot) \leftarrow$ truncate (new_entry$(\cdot), \omega$). This indicates any input data only affects at most ω rows in the truncated $\Delta\mathcal{V}$. Once the truncated outputs are available, Transform updates and re-shares the cardinality counter c, then appends the truncated outputs to the secure cache $\boldsymbol{\sigma}$ (Alg 1:5-7).

q-stable transformation. We now provide the following lemmas with respect to the q-stable transformation.

Lemma 1. (q-Stable Transformation [62, Definition 2]). *Let $T : \mathbb{D} \rightarrow \mathbb{D}$ to be a transformation, we say T is q-stable, if for any two databases $\mathcal{D}_1, \mathcal{D}_2 \in \mathbb{D}$, it satisfies $||T(\mathcal{D}_1) - T(\mathcal{D}_2)|| \leq q||\mathcal{D}_1 - \mathcal{D}_2||$*

Lemma 2. *Given T is q-stable T, and an ϵ-DP mechanism $\mathcal{M}$. The composite computation $\mathcal{M} \circ T$ implies $q\epsilon$-DP* [62, Theorem 2].

According to Lemma 1, it's clear that protocol Transform is q-stable, and thus by Lemma 2, applying an ϵ-DP mechanism over the outputs of Transform protocol (the cached data) implies $q\epsilon$-DP over the original data. Therefore, if q is constant, then the total privacy loss with respect to the input of Transform is bounded.

Contribution over time. According to the overall architecture, Transform is invoked repeatedly for transforming outsourced data into view entries at each time step. Thus having a q-stable Transform does not immediately imply bounded privacy loss with respect to the logical database. There are certain cases where one record may contribute multiple times over time as the input to Transform. For example, suppose the servers maintain a join table on both Alice's and Bob's data. When Alice submits new data, the servers need to compute new join tuples between her new data and Bob's entire database, which results in some of Bob's data being used multiple times. This could eventually lead to unbounded privacy loss.

Theorem 3. *Given a set of transformations $T = \{T_i\}_{i \geq 0}$, where each T_i is a q_i-stable transformation. Let $\{\mathcal{M}_i\}_{i \geq 0}$ be a set of mechanisms, where each $\mathcal{M}_i$ provides ϵ_i-differential privacy. Let another mechanism $\mathcal{M}(\mathcal{D})$ that executes each $\mathcal{M}_i$ using independent randomness with input $T_i(\mathcal{D})$. Then $\mathcal{M}$ satisfies ϵ-DP, where*

$$\epsilon = \max_{u,\mathcal{D}} \left(\sum_{i\,:\,\tau_i(u)>0} q_i \epsilon_i \right) \quad (4)$$

and $\tau_i(u) = ||T_i(\mathcal{D}) - T_i(\mathcal{D} - \{u\})||$, denotes the contribution of record u to the transformation T_i's outputs.

Theorem 3 shows that the overall privacy loss may still be infinite when applying the DP mechanisms over a composition of q-stable transformations (i.e. repeatedly invoke Transform). However, if the composed transformation T is also q-stable, then one can still obtain bounded privacy loss as $\max_{u,\mathcal{D}} \left(\sum_{i\,:\,\tau_i(u)>0} q_i \epsilon_i \right) \leq q \max(\epsilon_i)$. Inspired by this, the following steps could help to obtain fixed privacy loss over time: First we assign a total contribution budget b to each outsourced data $ds_i \in \mathcal{DS}$. As long as a record is used as input to Transform (regardless of whether it contributes to generating a real view entry), it is consumed with a fixed amount of budget (equal to the truncation limit ω). Then Transform keeps track of the available contribution budget for each record over time and ensures that only data with a remaining budget is used. According to this design, the "life time" contribution of each outsourced data to the materialized view object is bounded by b.

Implementation of trans_truncate **operator.** Naïvely, this operator can be implemented via two separate steps. For example, one may first apply an oblivious transformation (i.e. oblivious filter [7, 22], join [32, 89], etc.) without truncation over the input data. The results are stored to an exhaustively padded array. Next, truncation can be implemented by linearly scanning the array and resets the *isView* bit from 1 to 0 for a subset of the data in the array such that the resulting output satisfies Eq 3. In practice, truncation can be integrated with the view transformation so that the protocol does not have to run an extra round of linear scan. In what follows, we provide an instantiated implementation of oblivious sort-merge join where the output is truncated with a contribution bound b, and we continue to provide more implementations for other operators such as filter, nested-loop join, etc. in our complete full version.

Example 5.1 (b-truncated oblivious sort-merge join). *Assume two tables T_1, T_2 to be joined, the algorithm outputs the join table between T_1 and T_2 such that each data owns at most b rows in the resulting join table. The first step is to union the two tables and then obliviously sort [5] them based on the join attributes. To break the ties, we consider T_1 records are ordered before T_2 records. Then similar to a normal sort-merge join, where the operator linearly scans the sorted merged table then joins T_1 records with the corresponding records in T_2. There are some variations to ensure obliviousness and bounded contribution. First, the operator keeps track of the contribution of each individual tuple. If a tuple has already produced b join entries, then any subsequent joins with this tuple will be automatically discarded. Second, for linear scan, the operator outputs b entries after accessing each tuple in the merged table, regardless of how many true joins are generated. If there are fewer joins, then pad them with additional dummy data, otherwise truncate the true joins and keep only the b tuples. Figure 2 illustrates the aforementioned computation workflow.*

Secret-sharing inside MPC. When re-sharing the new cardinalities (Alg 1:5-6), we must ensure none of the two servers can tamper with or predict the randomness for generating secret shares. This can be done with the following approach: each outsourcing server $\mathcal{S}_i$ chooses a value z_i uniformly at random from the ring $\mathbb{Z}_{2^{32}}$, and contributes it as the input to Transform. The protocol then computes $[\![c]\!]^m \leftarrow \{c_0 \leftarrow z_0 \oplus z_1, c_1 \leftarrow c_0 \oplus c\}$ internally. By applying this, $\mathcal{S}_0$'s knowledge of the secret shared value is subject to the two

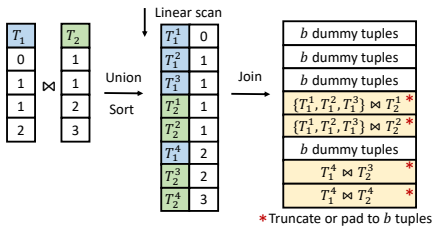


Figure 2: Oblivious truncated sort-merge join.

random values z_0, z_1 while $\mathcal{S}_1$'s knowledge is bounded to $c \oplus z_0$ which is masked with a random value unknown to $\mathcal{S}_1$. A complete security proof of this technique can be found in our full version.

5.2 Shrink Protocol

We propose two secure protocols that synchronize tuples from the secure cache to the materialized view. Our main idea originates from the *private synchronization strategy* proposed in DP-Sync [83], but with non-trivial variations and additional design. Typically, DP-Sync enforces trusted entities to execute private synchronization strategies, whereas in our scenario, the framework is supposed to allow untrusted servers to evaluate the view update protocol. Therefore, naïvely adopting their techniques could lead to additional leakage and exposure to untrusted servers, such as the internal states (i.e., randomness) during protocol execution. Furthermore, DP-Sync considers that the subjects evaluating the synchronization strategies have direct access to a local cache in the clear, whereas in our setup the protocol must synchronize cached view tuples from an exhaustively padded (with dummy entries) secure array without knowing how the real data is distributed. To address these problems, we incorporate the following techniques:

- We utilize a joint noise-adding mechanism to generate DP noise, which ensures that no admissible adversary can obtain or tamper with the randomness used to generate the noise.
- We implement a secure cache read operation that enforces the real data is always fetched before the dummy tuples when a cache read is performed.

In what follows, we review the technical details of two view update protocols, sDPTimer and sDPANT.

*5.2.1 Timer-based approach (*sDPTimer*).* sDPTimer is a 2PC protocol among $\mathcal{S}_0$, and $\mathcal{S}_1$, parameterized by T, ϵ and b, where it updates the materialized view every T time units with a batch of DP-sized tuples. Algorithm 2 shows the corresponding details.

Algorithm 2 sDPTimer

Input: ϵ; b (contribution budget); T (update interval); σ; $\mathcal{V}$;
1: **for** $t \leftarrow 1, \ldots$ **do**
2: **if** $t \mod T == 0$ **then**
3: recover c internally
4: $(z_0, z_1) \Leftarrow (\mathcal{S}_0, \mathcal{S}_1)$, s.t. $\forall z_i \xleftarrow{\text{rd}} \mathbb{Z}_{2^{32}}$
5: $z \leftarrow z_0 \oplus z_1$, $r \leftarrow$ fixed_point(z), s.t. $r \in (0, 1)$
6: $\text{sz} \leftarrow c + \frac{b}{\epsilon} \ln r \times \text{sign}(\text{msb}(z))$ ▷ $\text{sz} \leftarrow \text{Lap}(\frac{b}{\epsilon})$
7: $\hat{\sigma} \leftarrow \text{ObliSort}(\sigma, \text{key} = isView)$
8: $\mathbf{o} \leftarrow \hat{\sigma}[0, 1, 2, .., \text{sz} - 1]$, $\mathcal{V} \leftarrow \mathcal{V} \cup \mathbf{o}$, $\sigma \leftarrow \hat{\sigma}[\text{sz}, \ldots]$
9: reset $c = 0$ and re-share it to both servers.

For every T time steps sDPTimer obtains the secret-shared cardinality counter from both servers, and recovers the counter c internally. The protocol then distort this cardinality with Laplace noise sampled from $\text{Lap}(\frac{b}{\epsilon})$. To prevent information leakage, we must ensure none of the entities can control or predict the randomness used to generate this noise. To this end, inspired by the idea in [29], we implement a joint noise generation approach (Alg 2:4-6), where each server generates a random value $z_i \in \mathbb{Z}_{2^{32}}$ uniformly at random and contributes it as an input to sDPTimer. The protocol computes $z \leftarrow z_0 \oplus z_1$ internally, and converts z to a fixed-point random seed $r \in (0, 1)$. Finally, sDPTimer computes $\text{Lap}(\frac{b}{\epsilon}) \leftarrow \frac{b}{\epsilon} \ln r \times \text{sign}$, using one extra bit of randomness to determine the sign, i.e. the most-significant bit of z. By applying this, as long as one server honestly chooses the value uniformly at random and does not share it with any others (which is captured by the non-colluding server setting), she can be sure that no other entity can know anything about the resulting noise. In our design, this joint noise adding technique is used whenever the protocol involves DP noise generation. For ease of notation, we denote this approach as $\tilde{x} \leftarrow \text{JointNoise}(\mathcal{S}_0, \mathcal{S}_1, \Delta, \epsilon, x)$, where $\tilde{x} \leftarrow \text{Lap}(\frac{\Delta}{\epsilon})$.

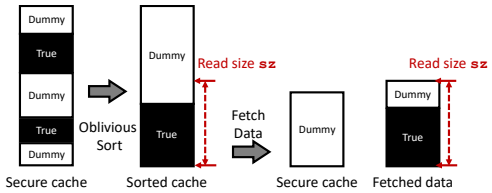


Figure 3: Cache read operation.

Next, sDPTimer obliviously sorts the exhaustively padded cache σ based on the $isView$ bit, moving all real tuples to the head and all dummy data to the tail, then cuts off the first sz elements from the sorted array and stores them as a separate structure $\mathbf{o}$. sDPTimer then updates the materialized view $\mathcal{V}$ by appending $\mathbf{o}$ to the old view instance. Figure 3 shows an example of the aforementioned operation. Such secure array operation ensures that the real data is always fetched before the dummy elements, which allows us to eliminate a subset of the dummy data and shrink the size of the updated view entries. Finally, after each update, sDPTimer resets c to 0 and re-shares it secretly to both participating servers.

Note that query errors of IncShrink are caused by two factors, namely, the valid data discarded by the Transform due to truncation constraints and the total amount of unsynchronized data in the cache. When the truncation bound is set too small, then a large number of valid view entries are dropped by Transform, resulting in inaccurate query answers. We further investigate how truncation bound would affect query errors in a later section (Section 7.4).

However, one could still choose a relatively large truncation bound to ensure that no data is discarded by Transform. As a result, query errors under such circumstances will be caused primarily by unsynchronized cached view tuples. Typically, less unsynchronized cached data that satisfy the requesting query implies better accuracy and vice versa. For instance, when IncShrink has just completed a new round of view update, the amount of cached data tends to be relatively small, and thus less data is missing from the materialized view. Queries issued at this time usually have better accuracy.

Theorem 4. *Given ϵ, b, and $k \geq 4 \log \frac{1}{\beta}$, where $\beta \in (0, 1)$. The # of deferred data c_d after k-th updates satisfies* $\Pr\left[c_d \geq \frac{2b}{\epsilon}\sqrt{k \log \frac{1}{\beta}}\right] \leq \beta$.

As per Theorem 4, we can derive the upper bound for total cached data at any time as $c^* + O(\frac{2b\sqrt{k}}{\epsilon})$, where c^* refers to the

number of newly cached entries since last update, and the second term captures the upper bound for deferred data. The data in the cache, although stored on the server, is not used to answer queries. Therefore large amounts of cached data can lead to inaccurate query results. Although one may also adopt strategies such as returning the entire cache to the client, or scanning the cache while processing the query so that no data is missing. This will not break the security promise, however, will either increase the communication cost or the runtime overhead as the secure cache is exhaustively padded with dummy data. Moreover, in later evaluations (Section 7.1), we observe that even without using the cache for query processing, the relative query errors are small (i.e. bounded by 0.04).

It's not hard to bound c^* by picking a smaller interval T, however the upper bound for deferred data accumulates when k increases. In addition, at each update, the server only fetches a batch of DP-sized data, leaving a large number of dummy tuples in the cache. Thus, to ensure bounded query errors and prevent the cache from growing too large, we apply an independent cache flushing mechanism to periodically clean the cache. To flush the cache, the protocol first sorts it, then fetches a set of data by cutting off a fixed number of tuples from the head of the sorted array. The fetched data is updated to the materialized view immediately and the remaining array is recycled (i.e. freeing the memory space). As per Theorem 4, we can set a proper flush size, such that with at most (a relatively small) probability β there is non-dummy data been recycled.

Theorem 5. *Given $\epsilon, b,$ and $k \geq 4 \log \frac{1}{\beta}$, where $\beta \in (0,1)$. Suppose the cache flush interval is f with size s. Then the number of data inserted to the view after the k-th update is bounded by $O(\frac{2b\sqrt{k}}{\epsilon}) + \frac{skT}{f}$.*

*5.2.2 Above noisy threshold. (*sDPANT*).* The above noise threshold protocol (Algorithm 3) takes θ, ϵ and b as parameters and updates the materialized view whenever the number of new view entries is approximately equal to a threshold θ.

Algorithm 3 sDPANT

Input: ϵ; b; θ (sync threshold); σ; $\mathcal{V}$;
1: $\epsilon_1 = \epsilon_2 = \frac{\epsilon}{2}$
2: $\tilde{\theta} \leftarrow \mathsf{JointNoise}(\mathcal{S}_0, \mathcal{S}_1, b, \epsilon_1/2, \theta)$ ▷ distort the threshold
3: $[\![\tilde{\theta}]\!]^m \leftarrow \mathsf{share}(\tilde{\theta}), [\![\tilde{\theta}]\!]^m \Rightarrow (\mathcal{S}_0, \mathcal{S}_1)$
4: **for** $t \leftarrow 1, \ldots$ **do**
5: recover c, and $\tilde{\theta}$ internally
6: $\tilde{c} \leftarrow \mathsf{JointNoise}(\mathcal{S}_0, \mathcal{S}_1, b, \epsilon_1/4, c)$
7: **if** $\tilde{c} \geq \tilde{\theta}$ **then** ▷ updates if greater than noisy threshold
8: $\mathsf{sz} \leftarrow \mathsf{JointNoise}(\mathcal{S}_0, \mathcal{S}_1, b, \epsilon_2, c)$
9: $\hat{\sigma} \leftarrow \mathsf{ObliSort}(\sigma, \mathsf{key} = isView)$
10: $\mathbf{o} \leftarrow \hat{\sigma}[0, 1, 2, .., \mathsf{sz} - 1], \mathcal{V} \leftarrow \mathcal{V} \cup \mathbf{o}, \sigma \leftarrow \hat{\sigma}[\mathsf{sz}, \ldots]$
11: $\tilde{\theta} \leftarrow \mathsf{JointNoise}(\mathcal{S}_0, \mathcal{S}_1, b, \epsilon_1/2, \theta)$
12: $[\![\tilde{\theta}]\!]^m \leftarrow \mathsf{share}(\tilde{\theta}), [\![\tilde{\theta}]\!]^m \Rightarrow (\mathcal{S}_0, \mathcal{S}_1)$
13: reset $c = 0$ and re-share it to both servers.

Initially, the protocol splits the overall privacy budget ϵ to two parts ϵ_1, and ϵ_2, where ϵ_1 is used to construct the noisy condition check (Alg 3:7) and ϵ_2 is used to distort the true cardinalities (Alg 3:8). The two servers then involve a joint noise adding protocol that securely distort θ with noise $\mathsf{Lap}(\frac{2b}{\epsilon_1})$. This noisy threshold will remain unchanged until sDPANT issues a new view update. An important requirement of this protocol is that such noisy threshold must remain hidden from untrusted entities. Therefore, to cache this value, sDPANT generates secret shares of $\tilde{\theta}$ internally and disseminates the corresponding shares to each server (Alg 3:3).

From then on, for each time step, the protocol gets the secret shares $[\![c]\!]^m$ (true cardinality) and $[\![\tilde{\theta}]\!]^m$ from $\mathcal{S}_0$ and $\mathcal{S}_1$, which are subsequently recovered inside the protocol. sDPANT distorts the recovered cardinality $\tilde{c} \leftarrow \mathsf{Lap}(\frac{4b}{\epsilon_1})$ and compares the noisy cardinality $\tilde{c}$ with $\tilde{\theta}$. A view update is posted if $\tilde{c} \geq \tilde{\theta}$. By issuing updates, sDPANT distorts c with another Laplace noise $\mathsf{Lap}(\frac{b}{\epsilon_2})$ to obtain the read size sz. Similar to sDPTimer, it obliviously sorts the secure cache and fetches as many tuples as specified by sz from the head of the sorted cache. Note that, each time when an update is posted, sDPANT must re-generate the noisy threshold with fresh randomness. Therefore, after each updates, sDPANT resets $c = 0$, produces a new $\tilde{\theta}$, and updates the corresponding secret shares stored on the two servers (Alg 3:11-13). In addition, the same cache flush method in sDPTimer can be adopted by sDPANT as well. The following theorem provides an upper bound on the cached data at each time, which can be used to determine the cache flush size.

Theorem 6. *Given $\epsilon, b,$ and let the cache flushes every f time steps with fixed flushing size s, the number of deferred data at any time t is bounded by $O(\frac{16b \log t}{\epsilon})$ and the total number of dummy data inserted to the materialized view is bounded by $O(\frac{16b \log t}{\epsilon}) + s\lfloor \frac{t}{f} \rfloor$.*

6 SECURITY PROOF

We provide the security and privacy proofs in this section.

Theorem 7. *IncShrink with* sDPTimer *satisfies Definition 4.*

Proof. We prove this theorem by first providing a mechanism $\mathcal{M}$ that simulates the update pattern leakage of the view update protocol and proving that $\mathcal{M}$ satisfies ϵ-DP. Second, we construct a *p.p.t* simulator that accepts as input only the output of $\mathcal{M}$ which can simulate the outputs that are computationally indistinguishable compared to the real protocol execution. In what follows, we provide the $\mathcal{M}_{\mathsf{timer}}$ that simulates the update pattern of sDPTimer.

$\mathcal{M}_{\mathsf{timer}}$ **return** $\mathsf{count}(\sigma_{t-T<t_{tid}}(\mathcal{D})) + \mathsf{Lap}(\frac{b}{\epsilon})$, **if** $0 \equiv t \pmod T$
$\forall\ t$: **return** 0, otherwise

where t_{tid} denotes the time stamp when tuple rid is inserted to $\mathcal{D}$, and $\sigma_{t-T<t_{tid}}$ is a filter operator that selects all tuples inserted within the time interval $(t-T, t]$. In general, $\mathcal{M}_{\mathsf{timer}}$ can be formulated as a series of $\frac{\epsilon}{b}$-DP Laplace mechanisms that applies over disjoint data (tuples inserted in non-overlapping intervals). Thus by parallel composition theorem [31], $\mathcal{M}_{\mathsf{timer}}$ satisfies $\frac{\epsilon}{b}$-DP. Moreover, by Lemma 2 given a q-stable transformation $\hat{T}$ such that $q = b$ (i.e. the Transform protocol), then $\mathcal{M}_{\mathsf{timer}}(\hat{T}(\mathcal{D}))$ achieves $b \times \frac{\epsilon}{b} = \epsilon$-DP. We abstract $\mathcal{M}_{\mathsf{timer}}$'s output as $\{(t, v_t)\}_{t \geq 0}$, where v_t is the number released by $\mathcal{M}_{\mathsf{timer}}$ at time t. Also we assume the following parameters are publicly available: $\epsilon, \mathbb{Z}_m, C_r$ (batch size of owner uploaded data), b (contribution bound), s (cache flush size), f (cache flush rate), T (view update interval). In what follows, we construct a *p.p.t.* simulator $\mathcal{S}$ that simulates the protocol execution with only access to $\mathcal{M}_{\mathsf{timer}}$'s outputs and public parameters (Table 1).

Initially, $\mathcal{S}$ initializes the internal storage B. Then for each time step, $\mathcal{S}$ randomly samples 2 batches of data B_1, B_2 from $\mathbb{Z}_m$, where the cardinality of B_1, and B_2 equals to C_r, and bC_r, respectively.

Simulator $\mathcal{S}(t, v_t, \epsilon, C_r, b, s, f, T)$		
1. Initialize internal storage $B \leftarrow \{\emptyset\}$.		
	i. $(B_1, B_2) \xleftarrow{rd} \mathbb{Z}_m$: $\lvert B_1\rvert = C_r, \lvert B_2\rvert = bC_r$	
	ii. $B_3 \leftarrow$ rd_fetch$(B_2 \cup B)$: $\lvert B_3\rvert = v_t$	
2. $\forall t > 0$	iii. **reveal** $(B_1, B_2, B_3, x \xleftarrow{rd} \mathbb{Z}_m)$ **if** $0 \equiv t \pmod T$	a. $B' \leftarrow$ rd_fetch(B) : $\lvert B'\rvert = s$
	iv. **if** $0 \equiv f \pmod t$	b. $B \leftarrow \{\emptyset\}$
		c. **reveal** $(B_1, B_2, B_3, B', x \xleftarrow{rd} \mathbb{Z}_m)$
	v. **reveal** (B_1, B_2, B_3) **otherwise**	

Table 1: Simulator construction (sDPTimer)

These two batches simulate the secret shared data uploaded by owners and the transformed tuples placed in cache at time t. Next, $\mathcal{S}$ appends B_2 to B and then samples B_3 from B such that the resulting cardinality of B_3 equals to v_t (if $v_t = 0$ then $\mathcal{S}$ sets $B_3 = \{\emptyset\}$). This step simulates the data synchronized by Shrink protocol. Finally, if $t \pmod T = 0$, $\mathcal{S}$ generates one additional random value x to simulate the secret share of the cardinality counter and reveals x together with the generated data batches to the adversary (2.iii). If $0 \equiv f \pmod t$, then $\mathcal{S}$ performs another random sample from internal storage B to fetch B' such that $|B'| = s$, followed by resetting B to empty. $\mathcal{S}$ reveals B_1, B_2, B_3, B' and one additional random value x to the adversary. These steps (2.iv) simulate the cache flush. Otherwise $\mathcal{S}$ only reveals B_1, B_2 and B_3. The computational indistinguishability between the B_1, B_2, B_3, x and the messages the adversary can obtain from the real protocol follows the security of $(2, 2)$-secret-sharing scheme and the security of secure 2PC [57]. □

Theorem 8. *IncShrink with* sDPANT *satisfies Definition 4.*

Proof. Following the same proof paradigm, we first provide $\mathcal{M}_{\text{ant}}$ that simulates the view update pattern under sDPANT.

$$
\begin{array}{ll}
 & \tilde{\theta} \leftarrow \theta + \text{Lap}(\frac{4b}{\epsilon}), \text{if } t = 0 \\
\mathcal{M}_{\text{ant}} & c_t \leftarrow \text{count}(\sigma_{t^* < t_{tid} \le t}(\mathcal{D})), \tilde{c}_t \leftarrow c_t + \text{Lap}(\frac{8b}{\epsilon}) \\
\forall\, t: & \textbf{return } c_t + \text{Lap}(\frac{4b}{\epsilon}), \theta + \text{Lap}(\frac{4b}{\epsilon}), \text{if } \tilde{c}_t \ge \tilde{\theta} \\
 & \textbf{return } 0, \text{if } \tilde{c}_t < \tilde{\theta}
\end{array}
$$

where $\sigma_{t^* < t_{tid} \le t}$ is a filter that selects all data received since last update. In general, $\mathcal{M}_{\text{ant}}$ is a mechanism utilizes sparse vector techniques (SVT) to periodically release a noisy count. According to [83] (Theorem 11), this mechanism satisfies $\frac{\epsilon}{b}$-DP (interested readers may refer to our full version for complete privacy proof of $\mathcal{M}_{\text{ant}}$). As per Lemma 2 we can obtain the same conclusion that $\mathcal{M}_{\text{ant}}(\hat{T}(\mathcal{D}))$ achieves ϵ-DP, if $\hat{T}$ is q-stable and $q = b$. Similarly, we abstract $\mathcal{M}_{\text{ant}}$'s output as $\{(t, v_t)\}_{t \ge 0}$ and we assume the following parameters are publicly available: ϵ, C_r, b, s, f, θ (threshold). For simulator construction one can reuse most of the components in Table 1 but with the following modifications: (i) For step 2.iii, one should replace the condition check as whether $v_t > 0$. (ii) For step 2.iii and 2.iv, the simulator outputs one additional random value $y \xleftarrow{rd} \mathbb{Z}_m$) which simulates the secret shares of refreshed noisy threshold. Similar, the indistinguishability follows the security property of XOR-based secret-sharing scheme.

□

7 EXPERIMENTS

In this section, we present evaluation results of our proposed framework. Specifically, we address the following questions:

- **Question-1**: Do the view-based query answering approaches have efficiency advantages over the non-materialization (NM) approach? Also, how does the DP-based view update protocol compare with the naïve ones?
- **Question-2**: For DP-protocols, is there a trade-off between privacy, efficiency and accuracy? Can we adjust the privacy parameters to achieve different efficiency or accuracy goals?
- **Question-3**: How do sDPTimer compare to the sDPANT? Under what circumstances is one better than the other one?

Implementation and configuration. We build the prototype IncShrink based on Shrinkwrap [7], a typical secure outsourced database under the server-aided MPC setting. Shrinkwrap only supports static data and a standard query answering mode. IncShrink extends it to a view-based SOGDB. In addition, we also implement client programs that consume data from a given dataset and outsource them to the server. This simulates how real-world data owner devices would receive and outsource new data. We implement all secure 2PC protocols using EMP-Toolkit-0.2.1 and conduct all experiments on the GCP instance with 3.8GHz Xeon CPU, 32Gb RAM.

Data. We evaluate the system using two datasets: *TPC Data Stream (TPC-ds)* [71], and *Chicago Police Database (CPDB)* [23]. TPC-ds collects the retail records for several product suppliers over a five-year period. In our evaluation, we mainly use two relational tables, the *Sales* and the *Return* table. After eliminating invalid data points with incomplete or missing values, the *Sales* and *Return* tables contain 2.2 million and 270,000 records, respectively. CPDB is a living repository of public data about Chicago's police officers and their interactions with the public. We primarily use two relations, the *Allegation* table, which documents the results of investigations into allegations of police misconduct, and the *Award* table, which collects awards given to certain officers. The cleaned data contains 206,000 and 656,000 records for *Allegation* and *Award*, respectively.

Execution scenario & Testing query. For TPC-ds data, we delegate each relational table to a client program, which then independently outsources the data to the servers. We multiplex the sales time (*Sales* table) or return time (*Return* table) associated with each data as an indication of when the client received it. In addition, we assume that the client program uploads a batch of data every single day and the uploaded data is populated to the maximum size. In addition, we pick the following query for evaluating with TPC-ds.

- Q1-Count the total number of products returned within 10 days after purchasing: "`SELECT COUNT(*) FROM Sales INNER JON Returns ON Sales.PID = Returns.PID WHERE Returns.ReturnDate - Sales.SaleDate <= 10`"

Moreover, we set the materialized view as a join table for all products that returned within 10 days. As Q1 has multiplicity 1, thus we set $\omega = 1$ (truncation bound), $b = 10$ (total contribution budget).

For CPDB data, we consider only *Allegation* table is private and is delegated to a client program. The *Award* table will be treated as a public relation. Again, we use the investigation case end time to indicate when the client program received this record, and we assume that the client outsource data once every 5 days (minimum time span is 5 days), and the data is padded to maximum possible size as well. For evaluation, we select the following query.

- Q2-Count how many times has an officer received an award from the department despite the fact that the officer had been found to have misconduct in the past 10 days: "`SELECT`

Comparison Cat.		DP-Timer	DP-ANT	OTM	EP	NM
Average query error						
TPCds	**L1 Error**	40.02	32.01	2008.92	0	0
	Relative Error	0.03	0.029	1	N/A	N/A
	Imp.†	**50×**	**63×**	**1×**‡	N/A	N/A
CPDB	**L1 Error**	61.93	52.45	6595.6	0	0
	Relative Error	0.043	0.038	1	N/A	N/A
	Imp.	**107×**	**126×**	**1×**	N/A	N/A
Average execution time (s)						
TPC-ds	Transform	9.72	9.69	N/A	9.71	N/A
	Shrink	0.34	0.37	N/A	N/A	N/A
	QET	0.051	0.052	0	5.84	7982
	Imp. (over NM)	**1.5e+5×**	**1.5e+5×**	N/A	1366×	**1×**
	Imp. (over EP)	**115×**	**112×**	N/A	**1×**	N/A
CPDB	Transform	2.93	2.91	0	2.93	N/A
	Shrink	3.93	3.77	N/A	N/A	N/A
	QET	0.17	0.17	0	51.36	1341
	Imp. (over NM)	**7888×**	**7888×**	N/A	26.1×	**1×**
	Imp. (over EP)	**302×**	**302×**	N/A	**1×**	N/A
Materialized view size (Mb)						
TPC-ds	**Avg. Size**	2.01	2.04	0.01	229.65	N/A
	Imp.	**114×**	**113×**	N/A	**1×**	N/A
CPDB	**Avg. Size**	6.63	6.68	0.01	2017.38	N/A
	Imp.	**304×**	**302×**	N/A	**1×**	N/A

† **Imp.** denotes the improvements; ‡ **1×** denotes the comparison baseline;

Table 2: Aggregated statistics for comparison experiments

```
COUNT(*) FROM Allegation INNER JON Award ON
Allegation.officerID = Award.officerID
WHERE Award.Time - Allegation.officerID <= 10".
```

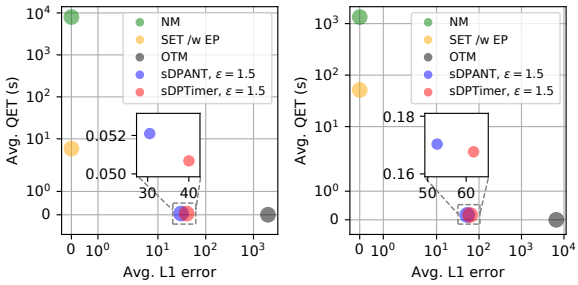
Similarly, the materialized view is a join table that process Q2. We set the truncation bound $\omega = 10$ and budget for each data as $b = 20$.

Default setting. Unless otherwise specified, we assume the following default configurations. For both DP protocols, we set the default privacy parameter $\epsilon = 1.5$, and cache flush parameters as $f = 2000$ (flush interval) and $s = 15$ (flush size). We fix the sDPANT threshold θ as 30 for evaluating both datasets. Since the average number of new view entries added at each time step is 2.7 and 9.8, respectively for TPC-ds and CPDB, thus for consistency purpose we set the timer T to $10 \leftarrow \lfloor \frac{30}{2.7} \rfloor$ and $3 \leftarrow \lfloor \frac{30}{9.8} \rfloor$. For each test group, we issue one test query at each time step and report the average L1 error and query execution time (QET) for all issued testing queries.

7.1 End-to-end Comparison

We address **Question-1** by performing a comparative analysis between the DP (sDPTimer, sDPANT), naïve (one-time materialization and exhaustive padding), and the non-materialization [83] approaches. The results are summarized in Table 2 and Figure 4.

Observation 1. View-based query answering provides a significant performance improvements over NM method. As per Table 2, we observe that the non-materialization method is the least efficient group among all groups. In terms of the average QET, the DP protocols achieve performance improvements of up to 1.5e+5× and 7888× on TPC-ds and CPDB data, respectively, in contrast to the NM approach. Even the EP method provides a performance edge of up to 1366× over NM approach. This result further demonstrates the necessity of adopting view-based query answering mechanism. A similar observation can be learned from Figure 4 as well, where in each figure we compare all test candidates along with the two dimensions of accuracy (x-axis) and efficiency (y-axis). In all figures, the view-based query answering groups lie beneath the NM approach, which indicates better performance.



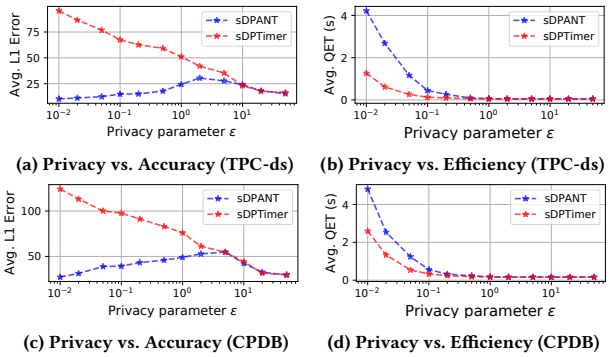
(a) TPC-ds group (b) CPDB group

Figure 4: End-to-end comparison.

Observation 2. DP protocols provide a balance between the two dimensions of accuracy and efficiency. According to Table 2, the DP protocols demonstrate at least 50× and 107× accuracy advantages (in terms of L1-error), respectively for TPC-ds and CPDB, over the OTM. Meanwhile, in terms of performance, the DP protocols show a significant improvement in contrast to the EP. For example, in TPC-ds group, the average QETs of both sDPTimer (0.051s) and sDPANT (0.052s) are almost 120× smaller than that of EP method (5.84s). Such performance advantage is even evident (up to 302×) over the CPDB data as testing query Q2 has join multiplicity greater than 1. Although, the DP approaches cannot achieve a complete accuracy guarantee, the average relative errors of all tested queries under DP protocols are below 4.3%. These results are sufficient to show that DP approaches do provide a balance between accuracy and efficiency. This conclusion can be better illustrated with Figure 4, where EP and OTM are located in the upper left and lower right corners of each plot, respectively, which indicates that they either completely sacrifice efficiency (EP) or accuracy (OTM) guarantees. However, both DP methods lie at the bottom-middle position of both figures, which further reveals that the DP protocols are optimized for the dual objectives of accuracy and efficiency.

7.2 3-Way Trade-off

We address **Question-2** by evaluating the DP protocols with different ϵ ranging from 0.01 to 50.



(a) Privacy vs. Accuracy (TPC-ds) (b) Privacy vs. Efficiency (TPC-ds)

(c) Privacy vs. Accuracy (CPDB) (d) Privacy vs. Efficiency (CPDB)

Figure 5: Trade-off experiment.

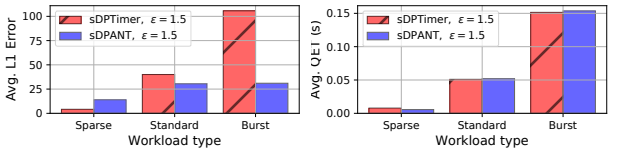
Observation 3. sDPTimer **and** sDPANT **exhibit different privacy-accuracy trade-off.** The accuracy-privacy trade-off evaluation is summarized in Figure 5a and 5c. In general, as ϵ increases from 0.01 to 50, we observe a consistent decreasing trend in the average L1 error for sDPTimer, while the mean L1 error for sDPANT first

increases and then decreases. According to our previous discussion, the error upper bound of sDPTimer is given by $c^* + O(\frac{2b\sqrt{k}}{\epsilon})$, where c^* denotes the cached new entries since last update, and $O(\frac{2b\sqrt{k}}{\epsilon})$ is the upper bound for the deferred data (Theorem 4). As sDPTimer has a fixed update frequency, thus c^* is independent of ϵ. However, the amount of deferred data is bounded by $O(\frac{2b\sqrt{k}}{\epsilon})$, which leads to a decreasing trend in the L1 error of sDPTimer as ϵ increases. On the other hand, the update frequency of sDPANT is variable and will be affected accordingly when ϵ changes. For example, a relatively small ϵ (large noise) will result in more frequent updates. As large noises can cause sDPANT to trigger an update early before enough data has been placed in the secure cache. As a result, a relatively small ϵ will lead to a correspondingly small c^*, which essentially produces smaller query errors. Additionally, when ϵ reaches a relatively large level, its effect on sDPANT's update frequency becomes less significant. Increasing ϵ does not affect c^* much, but causes a decrease in the amount of deferred data (bouned by $O(\frac{16\log t}{\epsilon})$ as shown in Theorem 6). This explains why there is a decreasing trend of sDPANT's L1 error after ϵ reaches a relatively large level. Nevertheless, both protocols show a privacy-accuracy trade-off, meaning that users can actually adjust privacy parameters to achieve their desired accuracy goals.

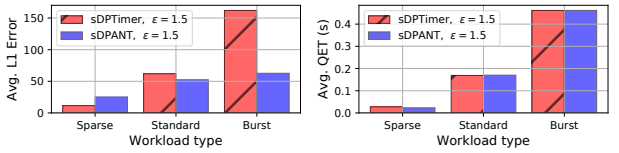
Observation 4. DP protocols have similar privacy-efficiency trade-off. Both DP protocols show similar trends in terms of efficiency metrics (Figure 5b and 5d), that is when ϵ increases, the QET decreases. It is because with a relatively large ϵ, the number of dummy data included in the view will be reduced, thus resulting in a subsequent improvement in query efficiency. As such, DP protocols also allow users to tune the privacy parameter ϵ in order to obtain their desired performance goals.

7.3 Comparison Between DP Protocols

We address **Question-3** by comparing the two DP protocols over different type of workloads. In addition to the standard one, for each dataset, we create two additional datasets. First, we sample data from the original data and create a *Sparse* one, where the total number of view entries is 10% of the standard one. Second, we process *Burst* data by adding data points to the original dataset, where the resulting data has 2× more view entries.



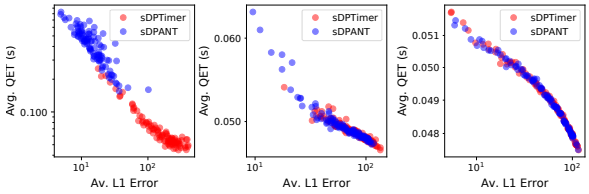
(a) Workload vs. Accuracy (TPC-ds) (b) Workload vs. Efficiency (TPC-ds)



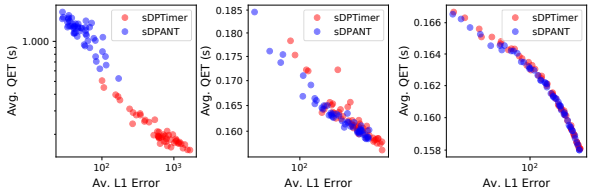
(c) Workload vs. Accuracy (CPDB) (d) Workload vs. Efficiency (CPDB)

Figure 6: DP protocols under different workloads.

Observation 5. sDPTimer **and** sDPANT **show accuracy advantages in processing** *Sparse* **and** *Burst* **data, respectively.** According to Figure 6, sDPTimer shows a relatively lower L1 error in the *Sparse* group than sDPANT. It is because it can take a relatively long time to have a new view entry when processing *Sparse* data. Applying sDPANT will cause some data to be left in the secure cache for a relatively long time. However, sDPTimer's update schedule is independent of the data workload type, so when the load becomes very sparse, the data will still be synchronized on time. This explains why sDPTimer shows a better accuracy guarantee against sDPANT for sparse data. On the contrary, when the data becomes very dense, i.e., there is a burst workload, the fixed update rate of sDPTimer causes a large amount of data to be stagnant in the secure cache. And thus causes significant degradation of the accuracy guarantee. However, sDPANT can adjust the update frequency according to the data type, i.e., the denser the data, the faster the update. This feature gives sDPANT an accuracy edge over sDPTimer when dealing with burst workloads. On the other hand, both methods show similar efficiency for all types of test datasets.



(a) TPC-ds, $\epsilon = 0.1$ (b) TPC-ds, $\epsilon = 1$ (c) TPC-ds, $\epsilon = 10$



(d) CPDB, $\epsilon = 0.1$ (e) CPDB, $\epsilon = 1$ (f) CPDB, $\epsilon = 10$

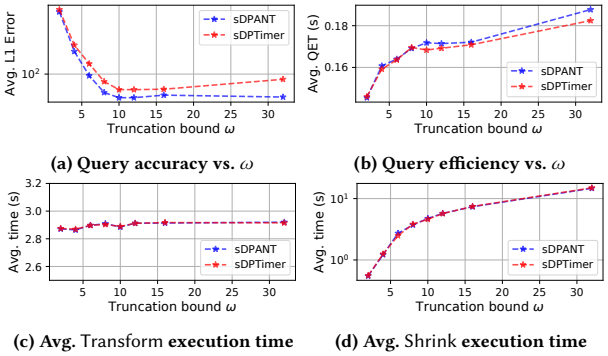
Figure 7: DP approaches under different workload.

Additionally, we also compare the two protocols with varying non-privacy parameters, i.e. T and θ, where we fix the ϵ then change T from 1-100, and correspondingly set θ according to T (As mentioned before, the average new view entries per moment are 2.7 and 9.8 for TPC-ds and CPDB data, respectively, thus we set θ to $3T$ and $10T$). We test the protocols with three privacy levels $\epsilon = 0.1, 1$ and 10 and report their comparison results in Figure 7.

Observation 6. When ϵ is small, two DP protocols have different biases in terms of accuracy and performance. According to Figure 7a and 7d, when $\epsilon = 0.1$, the data points for the sDPANT locate in the upper left corner of both figures, while the sDPTimer results fall on the opposite side, in the lower right corner. This implies that when ϵ is relatively small (privacy level is high), sDPANT tends to favor accuracy guarantees more, but at the expense of a certain level of efficiency. On the contrary, sDPTimer biases the efficiency guarantee. As per this observation, if users have strong demands regarding privacy and accuracy, then they should adopt sDPANT. However, if they have restrictive requirements for both privacy and performance, then sDPTimer is a better option. Moreover, the aforementioned deviations decrease when ϵ increases (Figure 7b). In addition, when ϵ reaches a relatively large value, i.e $\epsilon = 10$, both DP protocols essentially offer the same level of accuracy and efficiency guarantees.

7.4 Evaluation with Different ω

In this section, we investigate the effect of truncation bounds by evaluating IncShrink under different ω values. Since the multiplicity of Q1 is 1, the ω for answering Q1 is fixed to 1. Hence, in this evaluation, we focus on Q2 over the CPDB data. We pick different ω values from the range of 2 to 32 and set the contribution budget as $b = 2\omega$. The result is reported in Figure 8.



(a) Query accuracy vs. ω (b) Query efficiency vs. ω

(c) Avg. Transform execution time (d) Avg. Shrink execution time

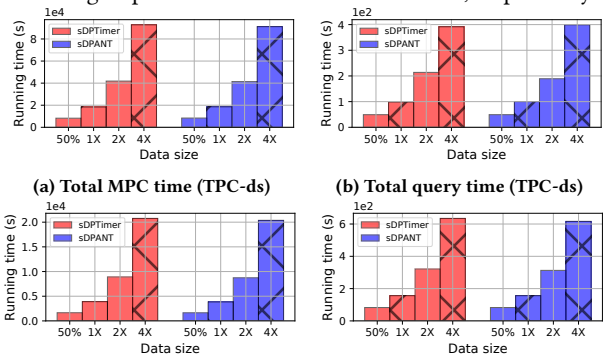
Figure 8: Evaluations with different truncation bound ω.

Observation 7. As ω grows from a small value, accuracy increases and efficiency decreases quickly. After ω reaches a relatively large value, sDPTimer **and** sDPANT **exhibit different trends in accuracy, but the same tendency in efficiency.** As per Figure 8, the average L1 error decreases when ω grows from $\omega = 2$. This is because when ω is small, many true view entries are dropped by the Transform protocol due to truncation constraint, which leads to larger L1 query errors. However, when ω reaches a relatively large value, i.e., greater than the maximum record contribution, then no real entries are discarded. At this point, increasing ω only leads to the growth of injected DP noises. As we have analyzed before, the accuracy under sDPANT can be better for relatively large noise, but the accuracy metric will be worse under sDPTimer method. On the other hand, dropping a large number of real entries (when ω is small) leads to a smaller materialized view, which consequently improves query efficiency. When ω is greater than the maximum record contribution, based on our analysis in Observation 4, keep increasing ω leads to both methods to introduce more dummy data to the view and causes its size to keep growing. As such, the efficiency continues decreasing.

Observation 8. The average Shrink **execution time increases along with the growth of ω, while the average execution time of** Transform **tends to be approximately the same.** The reason for this tendency is fairly straightforward. The execution time of both Transform and Shrink are dominated by the oblivious input sorting. The input size of Transform only relates to the size of data batches submitted by the users. Thus, changing ω does not affect the efficiency of Transform execution. However, the input size of Shrink is tied to ω, so as ω grows, the execution time increases.

7.5 Scaling Experiments

We continue to evaluate our framework with scaling experiments. To generate data with different scales, we randomly sample or replicate the original TPC-ds and CPDB data (We assign new primary key values to the replicated rows to prevent conflicts). According to Figure 9, for the largest dataset, i.e., the 4× groups, the total MPC time are around 24 and 6 hours, respectively for TPC-ds and CPDB. However, it is worth mentioning that for the 4× group, TPC-ds has 8.8 million and 1.08 million records in the two testing tables, and CPDB has 800K and 2.6 million records for *Allegation* and *Award* tables, respectively. This shows the practical scalability of our framework. In addition, the total query time for 4× TPC-ds and 4× CPDB groups are within 400 and 630 seconds, respectively.



(a) Total MPC time (TPC-ds) (b) Total query time (TPC-ds)

(c) Total MPC time (CPDB) (d) Total query time (CPDB)

Figure 9: Scaling experiments

8 EXTENSIONS

We discuss potential extensions of the original IncShrink design.

Connecting with DP-Sync. For ease of demonstration, in the prototype design, we assume that data owners submit a fixed amount of data at fixed intervals. However, IncShrink is not subject to this particular record synchronization strategy. Owners can choose other private update policies such as the ones proposed in DP-Sync, and can also adapt our framework. Additionally, the view update protocol requires no changes or recompilation as long as the view definition does not change. On the other hand, privacy will still be ensured under the composed system that connects IncShrink with DP-Sync. For example, assume the owner adopts a record synchronization strategy that ensures ϵ_1-DP and the server is deployed with IncShrink that guarantees ϵ_2-DP with respect to the owner's data. By sequential composition theorem [31], revealing their combined leakage ensures $(\epsilon_1 + \epsilon_2)$-DP over the owner's data. Similarly, such composability can also be obtained in terms of the accuracy guarantee. For instance, let's denote the error bound for the selected record synchronization policy as α_r (total number of records not uploaded in time). Then by Theorem 4 and 6, the combined system ensures error bounds $O(b\alpha_r + \frac{2b}{\epsilon}\sqrt{k})$ and $O(b\alpha_r + \frac{16b\log t}{\epsilon})$ under sDPTimer and sDPANT protocol, respectively.

Support for complex query workloads. We discuss how to generalize the view update protocol for complex query workloads, i.e. queries that can be written as a composite of multiple relational algebra operators. Apparently, one can directly replicate the design of this paper to support complex queries, by re-designing Transform protocol so that it computes the view tuples based on the specified query plan. However, there exists another design pattern that utilizes multi-level "Transform-and-Shrink" protocol. For example, we can disassemble a query into a series of operators and then construct an independent "Transform-and-Shrink" protocol for each individual operator. Moreover, the output of one "Transform-and-Shrink" protocol can be the input of another one, which eventually

forms a multi-level view update protocol. There are certain benefits of the multi-level design, for instance, one can optimize the system efficiency via operator level privacy allocation [7]. Recall that in Section 5.2 we discussed that the choice of privacy budget affects the number of dummy records processed by the system, with a higher proportion of dummy records reducing overall performance and vice versa. To maximize performance, one can construct an optimization problem that maximizes the efficiency of all operators in a given query, while maintaining the desired accuracy level. With a privacy budget allocation determined by the optimization problem, each operator can carry out its own instance of IncShrink, minimizing the overall computation cost while satisfying desired privacy and accuracy constraints. Note that optimization details are beyond the scope of this paper but may be of independent interest and we leave the design of these techniques to future work.

Expanding to multiple servers. In what follows, we summarize the major modifications that would extend our current design to a N servers setup such that $N \geq 2$. Firstly, the owners need to share their local data using the (N, N)-secret-sharing scheme, and disseminate one share per participating server. In addition, for all outsourced objects, such as the secure cache, the materialized view, and parameters passed between view update protocols, must be stored on the N servers in a secret shared manner. Secondly, both Transform and Shrink protocol will be compiled as a general MPC protocol where N parties (servers) provide their confidential input and evaluate the protocol altogether. Finally, when generating DP noises, each server needs to contribute a random bit string to the MPC protocol, which subsequently aggregates the N random strings to obtain the randomness used for noise generation. Note that our joint noise addition mechanism ensures to produce only one instance of DP noise, thus expanding to N servers setting does not lead to injecting more noise. According to [51, 52], such design can tolerate up to $N - 1$ server corruptions.

9 RELATED WORK

Secure outsourced database and leakage abuse attacks. There have been a series of efforts under the literature of secure outsourcing databases. Existing solutions utilize bucketization [40–42], predicate encryption [59, 75], property and order preserving encryption [2, 9, 12, 13, 66, 68, 69], symmetric searchable encryption (SSE) [3, 11, 20, 26, 35, 45, 46, 48, 50, 67, 78], functional encryption [15, 74], oblivious RAM [6, 24, 28, 43, 65, 89], multi-party secure computation (MPC) [6, 7, 14, 79], trusted execution environments (TEE) [32, 70, 81, 87] and homomorphic encryption [16, 22, 34, 72]. These designs differ in the types of supported queries and the provided security guarantees. Although the initial goal was to conceal the record values [2, 6, 9, 12, 13, 15, 40, 42, 59, 66, 69, 69, 75], researchers soon discovered the shortcomings of this security assurance. Recent works have revealed that these methods may be subject to certain leakage through query patterns [84, 88], access patterns [27, 49] and query response volume [37–39, 49], which makes them vulnerable to leakage-abuse attacks [10, 19]. Thus, more recent works on secure outsourced databases not only consider concealing record values but also hiding associated leakages [3, 6, 7, 11, 20, 24, 28, 32, 35, 43, 45, 46, 50, 65, 67, 78, 87, 89]. Unfortunately, few of the aforementioned efforts consider the potential leakage when data is dynamic [3, 20, 35, 50]. Wang et al. [83] formalize a general leakage named *update pattern* that may affect many existing secure databases when outsourcing dynamic data.

Differentially-private leakage. Existing studies on hiding database leakage with DP can be divided into two main categories: (i) safeguarding the query results from revealing sensitive information [1, 22, 25, 56, 60], and (ii) obscuring side-channel leakages such as access pattern [7, 21, 50, 61, 73, 82], query volume [11, 67] and update patterns [83]. The first category consists of works that enable DP query answering over securely provisioned (and potentially dynamic) data. Since these efforts typically focus solely on query outputs, side-channel leakages are not considered or assumed to be eliminable by existing techniques. Works in the second group focus on hiding side-channel information with DP, which is pertinent to our study. Among those, [7] and [83] are the two most relevant works to our study. [7] extends the work of [6], both of which use MPC as the main tool to architect secure outsourced databases. However, [6] fails to address some important leakages associated with intermediate computation results (i.e., the size of some intermediate outputs may leak sensitive information about the underlying data). Thus, [7] is proposed to fill this gap. [7] implements a similar resizing technique as IncShrink that ensures the volume leakage per secure operator is bounded by differential privacy, however, their system is restrictively focused on processing static data. [83] considers hiding update patterns when outsourcing growing data with private update strategies. However, they mandate that the update strategies must be enforced by trusted entities, while IncShrink allows untrusted servers to privately synchronize the materialized view. Additionally, [83] considers the standard mode that processes queries directly over outsourced data, which inevitably incurs additional performance overhead. Interested readers may refer to Sections 5.1 and 5.2, where we provide more in-depth comparisons between IncShrink and [7, 83], and highlight our technical contributions.

Bounding privacy loss. There is a series of work investigating approaches to constrain the privacy loss of queries or transformations with unbounded stability [44, 54, 55, 62, 80, 85]. However these works are conducted under the scope of standard databases rather than secure outsourced databases. Moreover, most of the works consider to bound the privacy loss of a single query or one-time transformation [44, 62, 80, 85]. In this work, we consider constraining the privacy loss of a composed transformation, which may contain an infinite number of sub-transformations.

10 CONCLUSION

In this paper, we present a framework IncShrink for outsourcing growing data on untrusted servers while retaining the query functionalities over the outsourced data. IncShrink not only supports an efficient view-based query answering paradigm but also ensures bounded leakage in the maintenance of materialized view. This is achieved by (i) utilizing incremental MPC and differential privacy to architect the view update protocol and (ii) imposing constraints on record contributions to the transformation of view instances.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grants 2029853, 2016393.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase.. In *CIDR*.

[5] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[6] Johes Bater, Satyender Goel, Gregory Elliott, Abel Kho, Craig Eggen, and Jennie Rogers. 2016. SMCQL: Secure querying for federated databases. *Proceedings of the VLDB Endowment* 10, 6 (2016), 673–684.

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

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

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

[11] 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).

[12] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. 2009. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 224–241.

[13] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*. Springer, 578–595.

[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] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[18] Yang Cao, Masatoshi Yoshikawa, Yonghui Xiao, and Li Xiong. 2017. Quantifying differential privacy under temporal correlations. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 821–832.

[19] 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.

[20] 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.

[21] 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.

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

[23] cpdp.co. 2006. Chicago Police Database. https://github.com/invinst/chicago-police-data

[24] 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

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

[26] 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.

[27] 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.

[28] 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)*.

[29] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 486–503.

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

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

[33] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 386–408.

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

[35] 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.

[36] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.

[37] 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.

[38] 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.

[39] 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.

[40] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 216–227.

[41] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure multidimensional range queries over outsourced data. *The VLDB Journal* 21, 3 (2012), 333–358.

[42] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. 2004. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 720–731.

[43] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private large-scale databases with distributed searchable symmetric encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 90–107.

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

[45] Seny Kamara and Tarik Moataz. 2018. SQL on structurally-encrypted databases. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 149–180.

[46] Seny Kamara and Tarik Moataz. 2019. Computationally volume-hiding structured encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 183–213.

[47] Seny Kamara, Payman Mohassel, and Mariana Raykova. 2011. Outsourcing Multi-Party Computation. *IACR Cryptol. Eprint Arch.* 2011 (2011), 272.

[48] 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.

[49] 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.

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

[51] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 158–189.

[52] Marcel Keller, Peter Scholl, and Nigel P Smart. 2013. An architecture for practical actively secure MPC with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 549–560.

[53] Daniel Kifer and Ashwin Machanavajjhala. 2011. No free lunch in data privacy. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 193–204.

[54] 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.

[55] Ios Kotsogiannis, Yuchao Tao, Ashwin Machanavajjhala, Gerome Miklau, and Michael Hay. 2019. Architecting a Differentially Private SQL Engine.. In *CIDR*.

[56] 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

[57] Yehuda Lindell. 2017. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography* (2017), 277–346.

[58] Changchang Liu, Supriyo Chakraborty, and Prateek Mittal. 2016. Dependence Makes You Vulnberable: Differential Privacy Under Dependent Tuples.. In *NDSS*, Vol. 16. 21–24.

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

[60] 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.

[61] 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.

[62] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 19–30.

[63] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational differential privacy. In *Annual International Cryptology Conference*. Springer, 126–142.

[64] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[65] 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.

[66] Omkant Pandey and Yannis Rouselakis. 2012. Property preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 375–391.

[67] 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.

[68] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* 2016 (2016), 591.

[69] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: processing queries on an encrypted database. *Commun. ACM* 55, 9 (2012), 103–111.

[70] 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.

[71] RELATIONAL DATASET REPOSITORY. [n.d.]. TPCDS. https://relational.fit.cvut.cz/dataset/TPCDS

[72] 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.

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

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

[75] 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.

[76] Shuang Song, Yizhen Wang, and Kamalika Chaudhuri. 2017. Pufferfish privacy mechanisms for correlated data. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1291–1306.

[77] Divesh Srivastava, Shaul Dar, Hosagrahar V Jagadish, and Alon Y Levy. 1996. Answering queries with aggregation using views. In *VLDB*, Vol. 96. 318–329.

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

[79] 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).

[80] Yuchao Tao, Xi He, Ashwin Machanavajjhala, and Sudeepa Roy. 2020. Computing Local Sensitivities of Counting Queries with Joins. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 479–494.

[81] 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.

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

[83] 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).

[84] 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.

[85] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially private sql with bounded user contribution. *Proceedings on privacy enhancing technologies* 2020, 2 (2020), 230–250.

[86] Yonghui Xiao and Li Xiong. 2015. Protecting locations with differential privacy under temporal correlations. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1298–1309.

[87] 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).

[88] 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.

[89] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 283–298.