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  
avin@cs.duke.edu

ABSTRACT
In this paper, we consider secure outsourced growing databases that support view-based query answering. These databases allow untrusted servers to privately maintain a materialized view, such that they can use only the materialized view to process query requests instead of accessing the original data from which the view was derived. To tackle this, we devise a novel view-based secure outsourced growing database 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 empirical evaluations 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 guarantees, and offers at least a 7,800× performance advantage over standard secure outsourced databases that do not support view-based query paradigm.

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 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
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 primary research goals in Section 2.1. Then we briefly review the framework components in Section 2.2. We provide a running example in Section 2.3 to illustrate the framework workflow as well as the overall architecture.

### 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’ logical data. As we 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. 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 cooperate 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 according to its internal states. To prevent the inclusion of a large amount of dummy data, Shrink shrinks the cached data into 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 differential privacy.

KI-3. Fixed privacy loss through constraints on record contributions. When IncShrink releases noisy cardinalities, it ensures $\epsilon$-DP with respect to the view instance. However, this does not imply $\epsilon$-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 over record contributions, IncShrink is still able to support a rich class of queries with relatively 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 $S_0$ and $S_1$. 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 but also ensures bounded leakage. More design and implementation details about the view update protocol 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 $\sigma$ 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 illustrate its workflow with a running example (as shown in 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 then 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 update functionality defined by the underlying database). For demonstration purposes, we assume that 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 join outputs will be padded to the maximum size then placed into a secure cache. Next, Shrink is executed independently, where it periodically synchronizes data from the secure cache to the materialized 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 to the real view tuples left in the secure cache 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
We write \( D \) of the growing database of (logical) updates, \( D \) Growing database. A growing database is a dynamic relational database. The theory of MPC offers strong security guarantee similar as what can be achieved with a trusted third party [56], i.e., absolutely no information leak to each participant \( P_i \) beyond the desired output of \( f(x_1, x_2, \ldots, x_n) \) 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 \( m = 2^t \), A \((n, t)\)-secret sharing \((t\text{-out-of-}n)\) 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 \( \left\lfloor x \right\rfloor^m \leftarrow (x_1, x_2, \ldots, x_n) \). There are many existing efforts to implement such secret sharing design [8], we focus on XOR-based \((2, 2)\)-secret sharing over \( \mathbb{Z}_{2^t} \) with the following specifications:

- **Generate shares** \( \text{share}(x) \): Given \( x \in \mathbb{Z}_m \), sample random values \( x_1 \leftarrow \mathbb{Z}_m \), compute \( x_2 \leftarrow x \oplus x_1 \), and return secret shares \( \left\lfloor x \right\rfloor^m \leftarrow (x_1, x_2) \).
- **Recover shares** \( \text{recover}(\left\lfloor x \right\rfloor^m) \): Given secret shares \( \left\lfloor x \right\rfloor^m \leftarrow (x_1, x_2) \), compute \( x \leftarrow x_1 \oplus x_2 \), then return \( x \).

## 3 Preliminaries

**Multi-party secure computation (MPC).** MPC utilizes cryptographic primitives to enable a set of participants \( P_1, P_2, \ldots, P_n \) to jointly compute a function \( f \) over private input data \( x \) 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 [56], i.e., absolutely no information leak to each participant \( P_i \) beyond the desired output of \( f(x_1, x_2, \ldots, x_n) \) and their input \( x_i \). In this work we focus mainly on the 2-party secure computing setting.

**Outsourced data.** The outsourced data is denoted as \( DS \), 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 \( DS = (DS_1 \cup \cdots \cup DS_t) \) at time \( t \).

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

**Query.** Given a growing database \( D \) and a corresponding materialized view \( V \), we define the logical query posted at time \( t \) as \( q_t(D) \) and the re-written view-based query as \( \tilde{q}_t(V) \). We refer the L1 norm of the difference between \( \tilde{q}_t(V) \) and \( q_t(D) \) as the L1 query error, denoted as \( L_{q_t} = ||\tilde{q}_t(V) - q_t(D)||_1 \), which measures the difference between the server responded outputs and their corresponding logical results. Additionally, we call the elapsed time for processing \( \tilde{q}_t(V) \) as the query execution time (QET) of \( q_t \).

**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 \( S_0 \) and \( S_1 \). We assume there exists a semi-honest **probabilistic polynomial time (p.p.t) adversary** \( A \) 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 \( A \) 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 (Sm-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, \( D = \{u_t\}_{t \geq 0} \), where \( u_t \) is a time stamped data. We write \( D = \{D_t\}_{t \geq 0} \), such that \( D_t \) denotes the database instance of the growing database \( D \) at time \( t \) and \( \forall D_t, D_t \subseteq D \).

**Outsourced data.** The outsourced data is denoted as \( DS \), 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 \( DS = (DS_1 \cup \cdots \cup DS_t) \) at time \( t \).

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

**Query.** Given a growing database \( D \) and a corresponding materialized view \( V \), we define the logical query posted at time \( t \) as \( q_t(D) \) and the re-written view-based query as \( \tilde{q}_t(V) \). We refer the L1 norm of the difference between \( \tilde{q}_t(V) \) and \( q_t(D) \) as the L1 query error, denoted as \( L_{q_t} = ||\tilde{q}_t(V) - q_t(D)||_1 \), which measures the difference between the server responded outputs and their corresponding logical results. Additionally, we call the elapsed time for processing \( \tilde{q}_t(V) \) as the query execution time (QET) of \( q_t \).

**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 \( D \), the update pattern for outsourcing \( D \) is the function family of UpdPatt(\( D \)) = \{UpdPatt_t(\( D \))\}_{t \geq 0}, with:

\[
\text{UpdPatt}_t(D) = (t, |T_t(D)|)
\]

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 \( D \). It may include information about the volume of the outsourced data and their corresponding insertion times. Moreover, if \( T_t(D) \leq D_t - D_{t-1} \), then this simply indicates the record insertion pattern [83].

**Definition 2 (Neighboring growing databases).** Given a pair of growing databases \( D \) and \( D' \), such that there exists some parameter \( \tau \geq 0 \), the following holds: (i) \( \forall t \leq \tau, D_t = D'_t \) (ii) \( \forall t > \tau, D_t \neq D'_t \) 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 \( \epsilon \text{-DP} \) if for any neighboring growing databases \( D \) and \( D' \), and any \( O \in \mathbb{O} \), where \( O \) is the universe of all possible outputs, it satisfies:

\[
\Pr \left[ F(D) \in O \right] \leq e^\epsilon \Pr \left[ F(D') \in O \right]
\]  

(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 \( u_t \in D \) as
a secret event, therefore such mechanism $F$ achieves $\epsilon$-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 $\epsilon$-user level DP can be achieved by setting the privacy parameter in Definition 3 to $\frac{\epsilon}{\ell}$, where $\ell$ 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 $\epsilon$-event DP algorithms provide an $\epsilon'$ bound on privacy loss when data are correlated. For certain correlations, $\epsilon'$ can be even close to $\epsilon$ 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 $\epsilon$, 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 $\Pi$ is said to satisfy $\epsilon$-SIM-CDP if there exists a p.p.t. simulator $S$ 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 $D$, and any p.p.t. adversary $\mathcal{A}$, the adversary’s advantage satisfies:

$$\Pr\left[\mathcal{A}\left(\text{VIEW}^D(D, pp) = 1\right)\right] \leq \Pr\left[\mathcal{A}\left(\text{VIEW}^S(F(D), pp) = 1\right)\right] + \negl(\kappa)$$

(2)

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

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 how much data is shared by differential privacy. In other words, any p.p.t. adversary’s knowledge of such protocol $\Pi$ is restricted to the outputs of an $\epsilon$-DP mechanism $F$. We refer to the mechanism $F$ as the leakage profile of protocol $\Pi$, and a function related to the update pattern, i.e., $F(D) = f(\text{UpdPat}(D))$. 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 5.1 and 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 adding 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 cardinals.

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.

Algorithm 1 Transform protocol

**Input:** $\omega$ (truncation bound); $D; \sigma$.
1. if $t == 0$ then
2. $\langle c \rangle^m \leftarrow z \in \mathbb{Z}_m, \ x \in \mathbb{Z}_n, \ c \in \{S_0, S_1\}$
3. $\Delta V \leftarrow \text{trans-truncate}(D; \omega)$
4. $\langle c \rangle^m \leftarrow (S_0, S_1), \ c \leftarrow \text{recover}(\langle c \rangle^m)$
5. $c \leftarrow c + \sum_{i \in \Delta V} \phi + \r
6. $\langle c \rangle^m \leftarrow \text{share}(c), \langle c \rangle^m \Rightarrow \{S_0, S_1\}$
7. $\sigma \leftarrow \sigma \| \Delta V$
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 new view tuples and truncate the contribution of each record at the same time. More specifically, we assume the output of this server is stored in an exhaustively padded secure array $ΔV$, where each $Δ'V(i)$ is a transformed tuple with an extra isView bit to indicate whether the corresponding tuple is a view entry (isView=true) or a dummy tuple (isView=false). Additionally, the operator ensures that

$$\forall d_{s1} \in D_{s1}, ||g^α(D_{s1}) - g^α(D_{s1} - \{d_{s1}\})|| \leq ω$$

(3)

where $g^α(·) ←$ truncate (new_entry(·), $\omega$). This indicates any input data only affects at most $\omega$ rows in the truncated $ΔV$. Once the truncated outputs are available, Transform updates and re-shares the cardinality counter $c$, then appends the truncated outputs to the secure cache $a$ (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 $T : D \rightarrow D$ to be a transformation, we say $T$ is $q$-stable, if for any two databases $D_1, D_2 \in D$, it satisfies $||T(D_1) - T(D_2)|| \leq q||D_1 - D_2||$.

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

According to Lemma 1, it’s clear that protocol Transform is $q$-stable, and thus by Lemma 2, applying an $\epsilon$-DP mechanism over the outputs of Transform protocol (the cached data) implies $\epsilon$-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 $\{M_i\}_{i \geq 0}$ be a set of mechanisms, where each $M_i$ provides $\epsilon_i$-differential privacy. Let another mechanism $M(D)$ that executes each $M_i$ using independent randomness with input $T_i(D)$. Then $M$ satisfies $\epsilon$-DP, where

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

(4)

and $\tau_i(u) = ||T_i(D) - T_i(D - \{u\})||$, 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 invoked Transform). However, if the composed transformation $T$ is also $q$-stable, then one can still obtain bounded privacy loss as $\max_{u \in D} \left( \sum_{i : T_i(u) > 0} q_i \epsilon_i \right) \leq q \max(\epsilon_i)$. 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 $d_{s1} \in DS$. 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 $\omega$). 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.** Naively, 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 instantiatied 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.

**Figure 2:** Oblivious truncated sort-merge join.

**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 $S_i$ chooses a value $z_i$ uniformly at random from the ring $\mathbb{Z}_{2^n}$, and
contributes it as the input to Transform. The protocol then computes $[c]^m \leftarrow (c_0 \oplus z_0, c_1 \oplus z_1)$ internally. By applying this, $S_0$’s knowledge of the secret shared value is subject to the two random values $z_0, z_1$ while $S_1$’s knowledge is bounded to $c \oplus z_0$ which is masked with a random value unknown to $S_1$. 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 internally, and converts it as an input to $sDPTimer$. A complete security proof of this technique can be found in our full version.

Algorithm 2 sDPTimer

| Input: $e$, $b$ (contribution budget); $T$ (update interval); $\sigma$, $\mathcal{V}$; |
|---|
| for $t \leftarrow 1, ..., T$ do |
| if $t \mod T \equiv 0$ then |
| recover $c$ internally |
| $(z_0, z_1) \leftarrow (S_0, S_1)$, s.t. $\forall z_i \leftarrow \mathcal{Z}_{2^k}$ |
| $z \leftarrow z_0 \oplus z_1$, $r \leftarrow \text{fixed_point}(z)$, s.t. $r \in (0, 1)$ |
| $sz \leftarrow c + z/\ln r \times \text{sign} \big(\text{msb}(z)\big)$ $\triangleright$ $sz \leftarrow \text{Lap}(b)$ |
| $\sigma \leftarrow \text{OblSort}(\sigma, \text{key} = \text{isView})$ |
| $o \leftarrow \sigma[0, 1, 2, ..., sz - 1]$, $\mathcal{V} \leftarrow \mathcal{V} \cup o, \sigma \leftarrow \hat{\sigma}[sz, ...]$ |
| 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 $\text{Lap}(b)$. 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 $z_i \in \mathcal{Z}_{2^k}$ 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 $\text{Lap}(z) \leftarrow z/\ln r \times \text{sign}$, 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 $\hat{x} \leftarrow \text{JointNoise}(S_0, S_1, \Delta, \epsilon, x)$, where $\hat{x} \leftarrow \text{Lap}(\frac{\epsilon}{c})$.

![Figure 3: Cache read operation.](image)

Next, sDPTimer obliviously sorts the exhaustively padded cache $\sigma$ 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 $o$. sDPTimer then updates the materialized view $\mathcal{V}$ by appending $o$ 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 $\epsilon, b$, and $k \geq 4 \log \beta$, where $\beta \in (0, 1)$. The number of deferred data $c_d$ after $k$-th updates satisfies $\Pr[c_d \geq \alpha] \leq \beta$, where $\alpha = \frac{\epsilon}{\epsilon} \sqrt{k \log \frac{1}{\beta}}$.

As per Theorem 4, we can derive the upper bound for total cached data at any time as $c^* + O\left(\frac{\epsilon}{\epsilon} \sqrt{k \log \frac{1}{\beta}}\right)$, 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 because 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 $e^c$ 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 $\beta$ there is no dummy data been recycled.

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

### 5.2.2 Above noisy threshold (sDPANT)

The above noise threshold protocol (Algorithm 3) takes $\theta$, $e$ and $b$ as parameters and updates the materialized view whenever the number of new view entries is approximately equal to a threshold $\theta$.

**Algorithm 3 sDPANT**

1. **Input:** $e$; $b$; $\theta$ (sync threshold); $\sigma$; $\mathcal{V}$;
2. $\mathcal{\tilde{\theta}} \leftarrow \text{JointNoise}(S_0, T_1, \frac{e_1}{2}, \theta) \triangleright$ distort the threshold
3. $\hat{\theta}^m \leftarrow \text{share}(\mathcal{\tilde{\theta}}), [\hat{\theta}^m] \Rightarrow (S_0, S_1)$
4. for $t = 1,...,\infty$ do
5. recover c, and $\tilde{\theta}$ internally
6. if $\hat{\tilde{\theta}} \geq \tilde{\theta}$ then $\triangleright$ updates if greater than noisy threshold
7. $sz \leftarrow \text{JointNoise}(S_0, S_1, b, e_2, c)$
8. $\sigma \leftarrow \text{ObliSort}(\sigma, key = isView)
9. $o \leftarrow [0, 1, 2, ..., sz - 1], \mathcal{\tilde{\theta}} \leftarrow \mathcal{\tilde{\theta}} \cup o, \sigma \leftarrow \sigma[sz, ...]
10. $\mathcal{\tilde{\theta}} \leftarrow \text{JointNoise}(S_0, S_1, b, e_2, \theta)
11. $\hat{\theta}^m \leftarrow \text{share}(\mathcal{\tilde{\theta}}), [\hat{\theta}^m] \Rightarrow (S_0, S_1)$
12. reset c = 0 and re-share it to both servers.

Initially, the protocol splits the overall privacy budget $e$ to two parts $e_1$ and $e_2$, where $e_1$ is used to construct the noisy condition check (Alg 3.7) and $e_2$ is used to distort the true cardinalities (Alg 3.8). The two servers then involve a joint noise adding protocol that securely distort $\theta$ with noise Lap$(\frac{b}{2\theta_1})$. 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 $[\hat{\mathcal{e}}]^m$ (true cardinality) and $[\hat{\mathcal{b}}]^m$ from $S_0$ and $S_1$, which are subsequently recovered inside the protocol. sDPANT distorts the recovered cardinality $\hat{\mathcal{e}} \leftarrow \text{Lap}(\frac{b}{r_1})$ and compares the noisy cardinality $\hat{\theta}$ with $\hat{\tilde{\theta}}$. A view update is posted if $\hat{\tilde{\theta}} \geq \hat{\theta}$. By issuing updates, sDPANT distorts $c$ with another Laplace noise Lap$(\frac{b}{r_2})$ 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 $\hat{\theta}$, 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 $e$, $b$, and let the cache flushes every $t$ steps with fixed flushing size $s$. The number of deferred data at any time $t$ is bounded by $O(\frac{16b \log t}{e})$ and the total number of dummy data inserted to the materialized view is bounded by $O(\frac{16b \log t}{e}) + s\lfloor \frac{t}{f} \rfloor$.

### 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 $M$ that simulates the update pattern leakage of the view update protocol and proving that $M$ satisfies $\varepsilon$-DP. Second, we construct a p.p.t. simulator that accepts as input only the output of $M$ which can simulate the outputs that are computationally indistinguishable compared to the real protocol execution. In what follows, we provide the $M_{\text{timer}}$ that simulates the update pattern of sDPTimer.

$M_{\text{timer}}$ return $\text{count}(\sigma_{1-T < t_{\text{id}}}(\mathcal{D})) + \text{Lap}(\frac{\varepsilon}{e}),$ if $0 \equiv t \pmod{T}$

$\forall t :$ return $0,$ otherwise

where $t_{\text{id}}$ denotes the time stamp when tuple $r_{\text{id}}$ is inserted to $\mathcal{D}$, and $\sigma_{1-T < t_{\text{id}}}$ is a filter operator that selects all tuples inserted within the time interval $(t - T, t)$. In general, $M_{\text{timer}}$ can be formulated as a series of $\frac{\varepsilon}{e}$-DP Laplace mechanisms that applies over disjoint data (tuples inserted in non-overlapping intervals). Thus by parallel composition theorem [31], $M_{\text{timer}}$ satisfies $\frac{\varepsilon}{e}$-DP. Moreover, by Lemma 2 given a q-stable transformation $T$ such that $q = b$ (i.e. the Transform protocol), then $M_{\text{timer}}(T)(\mathcal{D})$ achieves $b \times \frac{\varepsilon}{e} = \varepsilon$-DP. We abstract $M_{\text{timer}}$’s output as $(t, V)$ and $V$ is the number released by $M_{\text{timer}}$ at time $t$. Also we assume the following parameters are publicly available: $e$, $Z_m$, $C_r$ (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 $S$ that simulates the protocol execution with only access to $M_{\text{timer}}$’s outputs and public parameters (Table 1).

Initially, $S$ initializes the internal storage $B$. Then for each time step, $S$ randomly samples 2 batches of data $B_1$, $B_2$ from $Z_m$, where the cardinality of $B_1$, and $B_2$ equals to $C_r$ and $bC_r$, respectively. These two batches simulate the secret shared data uploaded by owners and the transformed tuples placed in cache at time $t$. Next,
InShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy

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 implement a prototype InShrink, and evaluate it with real-world datasets. We build the prototype InShrink based on Shrinkwrap [7], a typical secure outsourced database scheme under the server-aided MPC setting. Shrinkwrap only supports static data and a standard query answering method. InShrink extends it to a view-based SOGDB that handles growing data. In addition to the prototype InShrink, we also implement client programs that consume data from the given datasets and outsource them to the server, which simulate how real-world data owner devices would receive and outsource new data.

We implement all secure 2PC protocols using EMP-Toolkit-0.2.1 package and conduct all experiments on the GCP instance with 3.8GHz Xeon CPU, 32Gb RAM, and 64 bit Ubuntu 18.04.1 OS.

### 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, for TPC-ds, 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 information on awards given to certain officers. The cleaned data (after eliminating invalid entries) contains 206,000 and 656,000 records for Allegation and Award table, 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 JOIN Returns ON Sales.PID = Returns.PID WHERE Returns.ReturnDate - Sales.SaleDate <= 10"
**Table 2: Aggregated statistics for comparison experiments**

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

**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 $\theta$ 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, for consistency purpose we set the timer $T$ to 10 $\left\lfloor \frac{30}{2f} \right\rfloor$ and 3 $\left\lfloor \frac{10}{2f} \right\rfloor$.

**End-to-end Comparison**

We address **Question-1** by performing a comparative analysis between DP protocols, naive protocols (one-time materialization and exhaustive padding method), and the non-materialization approach (standard SOGDB model [83]). The comparison 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+5x and 7888x on TPC-ds and CPDB data, respectively, in contrast to the NM approach. Even the EP method provides a performance edge of up to 1366x 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.

**7.1 End-to-end Comparison**

We address **Question-1** by evaluating the DP protocols with different $\epsilon$ ranging from 0.01 to 50.

**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 50x and 107x accuracy advantages (in terms of L1-error), respectively for TPC-ds and CPDB, over the OTM method. Meanwhile, in terms of performance, the DP protocols show a significant improvement in contrast to the EP method. For example, in TPC-ds group, the average QETs of both sDPTimer (0.051s) and sDPANT (0.052s) are almost 120x smaller than that of EP method (5.84s). Such performance advantage is even evident (up to 302x) 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 we can observe that 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 $\epsilon$ ranging from 0.01 to 50.

**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 $\epsilon$ 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 $c^* + O\left(\frac{2b\sqrt{k}}{\epsilon}\right)$, where $c^*$ denotes the cached new entries since last update, and $O\left(\frac{2b\sqrt{k}}{\epsilon}\right)$ is the upper bound for the deferred data (Theorem 4). As sDPTimer has a fixed update frequency, thus $c^*$ is independent of $\epsilon$. However, the amount of deferred data is bounded by $O\left(\frac{2b\sqrt{k}}{\epsilon}\right)$, which leads to a decreasing trend in the L1 error of sDPTimer as $\epsilon$ increases. On the other hand, the update frequency of sDPANT
Figure 5: Trade-off experiment.

is variable and will be affected accordingly when \( \epsilon \) changes. For example, a relatively small \( \epsilon \) (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 \( \epsilon \) will lead to a correspondingly small \( \epsilon^* \), which essentially produces smaller query errors. Additionally, when \( \epsilon \) reaches a relatively large level, its effect on sDPANT’s update frequency becomes less significant. Increasing \( \epsilon \) does not affect \( \epsilon^* \) much, but causes a decrease in the amount of deferred data (bounded by \( O\left(\frac{16\log T}{\epsilon^2}\right) \) as shown in Theorem 6). This explains why there is a decreasing trend of sDPANT’s L1 error after \( \epsilon \) 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 \( \epsilon \) increases, the QET decreases. It is because with a relatively large \( \epsilon \), the number of dummy data included in the view will be reduced, thus resulting in a subsequent improvement in query efficiency. Thus, similar to the accuracy-privacy trade-off, the DP protocols also provide a privacy-efficiency trade-off that allows users to tune the privacy parameter \( \epsilon \) 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.

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.

Figure 6: DP protocols under different workloads.

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.

Figure 7: DP approaches under different workload.

Additionally, we also compare the two protocols with varying non-privacy parameters, i.e. \( T \) and \( \theta \), where we fix the \( \epsilon \) then change \( T \) from 1-100, and correspondingly set \( \theta \) 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 \( \theta \) to 3T and 10T). We test the protocols with three privacy levels \( \epsilon = 0.1, 1, 10 \) and 10 report their comparison results in Figure 7.

Observation 6. When \( \epsilon \) 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 $\epsilon$ 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 $\epsilon$ increases (Figure 7b). In addition, when $\epsilon$ reaches a relatively large value, i.e., $\epsilon = 10$, both DP protocols essentially offer the same level of accuracy and efficiency guarantees. For example, for each “red” point in Figures 7c and 7f, one can always find a comparative “blue” dot.

7.4 Evaluation with Different $\omega$

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

![Figure 8: Evaluations with different truncation bound $\omega$.](image)

Observation 8. The average Shrink execution time increases along with the growth of $\omega$, 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 protocols is dominated by the oblivious input sorting. The input size of the Transform protocol is only related to the size of data batches submitted by the users. Therefore, changing $\omega$ does not affect the efficiency of Transform execution. However, the input size of Shrink is tied to $\omega$, so as $\omega$ grows, the execution time of Shrink 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 $4\times$ 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 $4\times$ 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 $4\times$ TPC-ds and $4\times$ CPDB groups are within 400 and 630 seconds, respectively.

![Figure 9: Scaling experiments](image)

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 $\epsilon_1$-DP and the server is deployed with IncShrink that guarantees $\epsilon_2$-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 $a_\tau$ (total number of records not uploaded in time). Then by Theorem 4 and 6, the combined system ensures error bounds $O(b_{\alpha} + \frac{b_{\beta}}{\tau}\sqrt{K})$ and $O(b_{\alpha} + \frac{b_{\beta} \log t}{\tau})$ under sDPTimer and sDPANT protocol, respectively. Interested readers may refer to our full version for the complete utility proofs.

Support for complex query workloads. Now we describe 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 first compiling a Transform protocol that produces and caches the corresponding view tuples based on the specified query plan, while a Shrink protocol is used independently to continuously synchronize the cached data. 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. Although in our prototype IncShrink design we assume to leverage 2 non-colluding servers, the system architecture can be modified to work with 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, 75], researchers soon discovered the shortcomings of this security assurance. Recent work has 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]. Therefore, 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 underlying data is dynamic [3, 20, 35, 50]. Wang et al. [83] formalize a general leakage named update pattern that may affect many existing secure database schemes 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 have presented a framework IncShrink for outsourcing growing data onto untrusted servers while retaining the privacy 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 secure view update protocol and (ii) imposing constraints on record contributions to the transformation of materialized view instance.

**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, Raman Krishnan 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 Workshops on Systems Security, 1–9.

[4] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghab Kaushal, Donald Kossmann, and Michelle购. 2016. SMQCL: Secure querying for federated databases. Proceedings of the VLDB Endowment 10, 6 (2016), 673–684.

[5] 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.

[6] Kenneth E. Batcher. 1968. Sorting networks and their applications. In The fourtieth annual ACM symposium on Theory of computing. 307–314.

[7] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 392–413.

[8] Avindar Chand, 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.

[9] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N Rothblum. 2010. Differential privacy. Foundations and Trends in Theoretical Computer Science 9, 3-4 (2014), 211–407.

[10] 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. 1038–1055.

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

[12] Dan Boneh, Europ. 2017. Evaluating 2-DNF formulas on ciphertexts. In Theory of cryptography conference. Springer, 325–341.

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

[14] 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.

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

[16] David Cash, Joseph Jaeger, Stanislav 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.

[17] 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.

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

[19] Kpydp.co. 2006. Chicago Police Database. https://github.com/invisitt/chicago-police-data.

[20] Natacha Crooys, Matthew Burke, Ethan Cecchetti, Sitlare 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, 723–743. https://www.usenix.org/legacy/usenixasia18/tech/abuse-attacks.

[21] Rachel Cummings, Sara Krehbiel, Kevin A Lai, and Uthaipong Tantipongpipat. 2018. Differential privacy for growing databases. arXiv preprint arXiv:1803.04616 (2018).

[22] 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.

[23] Jonathan L Dauwstr 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.

[24] Joanna 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).

[25] 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.

[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] 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. 1038–1055.

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

[29] 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. 1038–1055.
2002 ACM SIGMOD international conference on Management of data: 216–227.

Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcıoğlu. 2012. Secure multidimensional range queries over outsourced data. The VLDB Journal 21, 3 (2012), 335–358.

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. VLDB ’04, 729–739.

Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2006. Private large-scale databases with distributed searchable symmetric encryption. In Cryptographers’ Track at the RSA Conference. Springer, 90–107.

Noah Johnson, Joseph P. Neuf, and Dawn Song. 2011. Towards practical differential privacy for SQL queries. Proceedings of the VLDB Endowment 11, 5 (2018), 526–539.

Seny Kamara and Tarik Moataz. 2018. SQL on structurally-encrypted databases. In International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 149–180.

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.

Seny Kamara, Payman Mohassel, and Marianna Raykova. 2011. Outsourcing Multi-Party Computation. IACR Cryptol. Eprint Arch. 2011, 272.

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.

Georgios Kellaris, George Kolious, Kobi 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.

Georgios Kellaris, George Kolious, Kobi Nissim, and Adam O’Neill. 2017. Accessing data while preserving privacy. arXiv preprint arXiv:1706.01552 (2017).

Marcel Keller, Valerio Fazio, and Dragos Rotaru. 2018. Overdrive: Making SFPDZ great again. In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 158–189.

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 and communications security. 549–560.

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.

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.

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

Mathias Lécuyer, Raphael Slovin, Kiran Vodrahalli, Roxana Geambasu, and Daniel Htu. 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

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

Changchung Liu, Supriyo Chakraborty, and Prateek Mittal. 2018. Dependence on Operating Systems Design and Implementation. 490–507.

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.

Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2019. Computational differential privacy. In Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 375–391.

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.

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.

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

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.

Rishabh Poddar, Tobias Boelett, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. IACR Cryptol. Eprint Arch. 2016, 591.

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

Christian Pribe, Kapil Vaarni, and Manuel Costa. 2018. EnclaveDB: A secure database using SOX. In 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 264–278.

RELATIONAL DATASET REPOSITORY. [n.d.] TCPSD. https://relational.fit.cvut.cz/datatset/TCPSD

Ehabar Kuma 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.

Zhuwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. 2021. Oblu: Fuzzy access and search patterns in searchable encryption. arXiv preprint arXiv:2102.09651 (2021).

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

Elaine Shi, John Bethencourt, Hubert Chan, Dawn Song, and Adriano Perrig. 2007. Multi-dimensional range query over encrypted data. In 2007 IEEE Symposium on Security and Privacy (SP’07). IEEE, 350–364.

Shuang Song, Yirhen 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.

Divesh Srivastava, Shaad Dar, Hossaghrad V Jagadish, and Alon Y Levy. 1996. Answering queries with aggregation using views. In VLDB. Vol. 96. 318–329.

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

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

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.

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.

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

Chenghong Wang, Jokes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. DP-Sync: Hiding Update Patterns in Secure Outsourced Databases with Differential Privacy. arXiv preprint arXiv:2103.15942 (2021).

Yonghui Xiong, and Li Xiong. 2019. Periodic locating with differential privacy under temporal correlations. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 1298–1309.

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).

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.

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 ({{USENIX}} DS’17). 283–298.

A ADDITIONAL IMPLEMENTATION DETAILS

A.1 Truncated view transformation

In addition to the obvious sort-merge join discussed in Example 5.1, we continue to provide two additional instantiations of the truncated view transformation, namely the truncated selection and truncated nested loop join.
A.1.1 Oblivious selection (filter). Since each input record can only contribute to the output of the selection operator at most once. Therefore, it does not require us to have additional implementations to constraint the record contributions. To ensure obliviousness, the operator will return all input data as the output. However, only records that satisfy the selection predicated will have its isView bit set to 1. As a result, records that do not satisfy the selection predicate are treated as dummy tuples (isView=0).

A.1.2 Truncated (oblivious) nested-loop join. The truncated nested-loop join is similar to a normal nested-loop join, where the operator scans the first table (outer table), say \( T_1 \), and joins each of tuples in \( T_1 \) to the rows in the second table \( T_2 \) (inner table). However, additional operations are required to ensure obliviousness and the bounded record contribution. Algorithm 4 illustrates the details of this truncated nested-loop join method.

Algorithm 4 Transform protocol

```
Input: Two tables \( T_1 \) and \( T_2 \); Truncation bound \( b \)
1: assign_budget\((T_1 ∪ T_2, b)\)
2: \( o \leftarrow \text{init\_secure\_array}\()\)
3: for \( tupp_1 \in T_1 \) do
4: \( o_i \leftarrow \text{init\_secure\_array}\()\)
5: for \( tupp_2 \in T_2 \) do
6: \( \text{if budget}(tupp_1) > 0 \) and \( \text{budget}(tupp_2) > 0 \) then
7: \( \text{if } \text{tupp}_1.\text{key} == \text{tupp}_2.\text{key} \text{then} \)
8: \( o_i\text{append}(tupp_1.\text{tupp}_2.\text{isView} = 1) \)
9: \( \text{consume\_budget}(tupp_1, tupp_2, 1) \)
else
10: \( o_i\text{append}(\text{dummy}) \)
11: \( o_i \leftarrow \text{Oblisort}(o_i) \)
12: \( o_i \leftarrow o_i[0, 1, 2, ..., b - 1] \)
13: \( o \leftarrow o_1 \)
```

Initially, the operator assigns a contribution budget to each tuple in \( T_1 \) and \( T_2 \). This can be achieved by appending a fixed point number (i.e., 32-bit) after each tuple. Then for each tuple \( tupp_1 \in T_1 \), the operator joins with each tuple \( tupp_2 \in T_2 \). A join tuple is generated if and only if (i) both \( tupp_1 \) and \( tupp_2 \) have remaining budgets (Algo 4:6) and (ii) the two tuples share the same join key. Once a join tuple is generated, the operator consumes the budgets of both join tuples, subtracting their remaining budgets by one (Algo 4:9). Otherwise it generates a dummy tuple. Additionally, at the end of each inner loop (Algo 4:12-13), the operator obliviously sorts the intermediate tuples \( o_i \) and picks only the first \( b \) tuples stored in \( o_i \). Since the contribution bound is \( b \), thus the total number of true joins in \( o_i \) must not exceed \( b \). And by applying these steps (Algo 4:12-13) could help to reduce the cache I/O burden.

A.2 Generating secret shares inside MPC

We provide an implementation example on how to generate \( k \)-out-of-\( k \) XOR secret shares inside an MPC protocol follow by a sketch proof for it’s security.

We assume there are \( k \) participating owners \( P_1, P_2, ..., P_k \), and there is a secret value \( c \) that is computed inside MPC. We assume the secret share scheme is over the ring \( \mathbb{Z}_m \) and we denote the MPC protocol that computes \( c \) internally as protocol.

1. \( \forall P_i \), randomly samples \( k - 1 \) values \((z_1', z_2', ..., z_{k-1}') \) \( \text{rd} \) \( \mathbb{Z}_m \), from ring \( \mathbb{Z}_m \).
2. Each \( P_i \) inputs the sampled values to protocol.
3. \( c \) is computed inside protocol.
4. Before revealing outputs, the protocol computes \( \forall z_i', z_i^{\text{trunc}} \leftarrow z_i' \oplus z_j' \oplus ... \oplus z_k' \), internally.
5. The protocol computes secret shares \((x_1, x_2, ..., x_k)\), such that \( \forall j < (k - 1), x_j \leftarrow z_i' \), and \( x_k \leftarrow c \oplus z_1' \oplus z_2' \oplus ... \oplus z_{k-1}' \).
6. The protocol reveals one secret share \( x_1 \) to one participating owner.

Availability. For all shares \( \langle x \rangle^m \leftarrow (x_1, x_2, ..., x_k) \),
\[ \Pr[\text{recover} \langle x \rangle^m = c] = \Pr[x_1 \oplus x_2 \oplus ... \oplus x_k = c] = 1 \]

Confidentiality. We introduce the following lemma that defines the security of secret sharing with adversary.

Lemma 9. \( A \)-out-of-\( n \) secret sharing scheme (share, recover) over ring \( \mathbb{Z}_m \) is perfectly secure if for any adversary \( \mathcal{A} \), \( \forall S \subseteq \{1, 2, ..., n\} \) such that \( |S| < t \), and for any two messages \( m, m' \), the following holds:
\[ \Pr[\mathcal{A}(x_i | i \in S) = 1 : (x_1, x_2, ..., x_k) \leftrightarrow \text{share}(m)] = \Pr[\mathcal{A}(x'_i | i \in S) = 1 : (x'_1, x'_2, ..., x'_k) \leftrightarrow \text{share}(m')] \]

In our setting, we consider the adversary is able to obtain up to \( k - 1 \) out of \( k \) secret shares, and we prove the security by illustrating our implemented approach satisfies Lemma 9.

Let \( X(m) \) be the secret shares obtained by the adversary, \( \mathcal{A} \), and let \( X'(m) \) be the secret shares such that \( X' \subseteq \text{share}(m) \) and \( X'(m) \cap X'(m) = \emptyset \). Since \( \mathcal{A} \) is able to control \( k - 1 \) parties, thus we consider the following two cases: (i) if \( m \oplus z_1' \oplus ... \oplus z_{k-1}' \neq X'(m) \), then \( X'(m) \) is independent from input message \( m \), therefore \( \mathcal{A} \) cannot distinguish the shares for two different messages. (ii) if \( m \oplus z_1' \oplus ... \oplus z_{k-1}' \neq X'(m) \), then \( X'(m) \neq \emptyset \), and \( \exists z^* \neq (X \setminus m) \). Since \( z^* \) uses randomness that can not be controlled by \( \mathcal{A} \), thus \( \Pr[\mathcal{A}(z^*) = 1] \geq \Pr[\mathcal{A}(z^*) = 1] \). Moreover, as \( m \oplus z_1' \oplus ... \oplus z_{k-1}' \neq X'(m) \), since for any two messages, the adversary \( \mathcal{A} \) cannot distinguish between their secret shares.

B PROOF OF THEOREMS

B.1 Proof of Theorem 3

Proof. Assume two neighboring databases \( D, D' \), differ by one record \( u \). Let \( o = \{o_j\}_{j\geq0} \) and \( o' = \{o'_k\}_{k\geq0} \) be to the output of \( M(D) \) and \( M(D') \), respectively, where \( o_j \) and \( o'_k \) denotes \( M_j \)'s output. We use \( T(U_i) \) and \( T(U'_i) \) to denote the corresponding input of \( M_j \) when \( M \) applies over \( D \) and \( D' \), respectively. We know that \( \epsilon = \ln \left( \frac{\Pr[M(D)=o]}{\Pr[M(D')=o']} \right) = \ln \left( \prod_{i=1}^{r_1} \frac{\Pr[M_i(U_i)=o_i]}{\Pr[M_i(U'_i)=o'_i]} \right) \), therefore we obtain \( \epsilon \leq \max_u \ln \left( \prod_{i=1}^{r_1} \frac{\Pr[M_i(U_i)=o_i]}{\Pr[M_i(U'_i)=o'_i]} \right) \leq \Delta \leq 4 \). □

B.2 Proof of Theorem 4, 5

Lemma 10. Given \( k \) independent and identically distributed Laplace random variables, \( Y_1, Y_2, ..., Y_k \), where each \( Y_i \) is sampled from the
distribution \( \text{Lap}(\frac{\Delta}{2}) \), where \( \Delta \) denotes the sensitivity. Let \( Y = \sum_{i=1}^{k} Y_i \), and \( 0 < \alpha \leq k \frac{\Delta}{2} \), the following inequality holds

\[
\Pr \left[ Y \geq \alpha \right] \leq e^{-\alpha^2 / \beta}.
\]

**Proof.** The complete proof of Lemma 10 can be found in the Appendix C.1 of [83] and in [31]. \( \square \)

**Corollary 11.** Given \( k \) independent and identically distributed Laplace random variables, \( Y_1, Y_2, \ldots, Y_k \), where each \( Y_i \) is sampled from the distribution \( \text{Lap}(\frac{\Delta}{2}) \). Let \( Y = \sum_{i=1}^{k} Y_i \), and \( \beta \in (0, 1) \), the following inequality holds

\[
\Pr \left[ Y \geq \frac{\Delta}{2} \sqrt{k \log \frac{1}{\beta}} \right] \leq \beta.
\]

**Proof.** Continue with Lemma 10, let \( e^{-\alpha^2 / \beta} = \beta \), then \( \alpha = 2 \Delta \sqrt{k \log \frac{1}{\beta}} \), when \( k > 4 \log \frac{1}{\beta} \) the corollary holds. \( \square \)

**Proof.** (Theorem 4). Let \( \tilde{c}_k \) denotes the total number of records synchronized by Shrink protocol after \( k \) times update (without cache flush), and let \( c_k \) denotes the true cardinality of materialized view after \( k \) updates at time \( kt \). Knowing that \( \tilde{c}_k \leftarrow c_k + \sum_{i=1}^{k} Y_i \), where each \( Y_i \) is an i.i.d Laplace random variable drawn from the distribution \( \text{Lap}(\frac{\Delta}{2}) \), where \( b \) is the contribution upper bound for each record. Given \( \beta \in (0, 1) \), according to Corollary 11 we obtain that \( \Pr \left[ c_k - \tilde{c}_k \geq \alpha \right] \leq \beta \), such that \( \alpha = \frac{2 \Delta \sqrt{k \log \frac{1}{\beta}}}{b} \). Knowing that \( c_k - \tilde{c}_k \) computes the total number of records delayed after \( k \)-th updates, thus the theorem holds. \( \square \)

**Proof.** (Theorem 5). Similarly, let \( \tilde{c}_k \) denotes the total number of records synchronized by Shrink protocol after \( k \) times update, and let \( c_k \) denotes the true cardinality of materialized view after \( k \) updates (at time \( kt \)). When considering cache flush, we compute \( \tilde{c}_k \leftarrow c_k + \sum_{i=1}^{k} Y_i + \sum_{j'=1}^{k'} s_j \), where \( k' \) denotes the number of cache flushes occurred since \( t = 0 \) and \( s \) is the cache flush size.

Knowing that \( \tilde{c}_k \leftarrow c_k + \sum_{i=1}^{k} Y_i \), where each \( Y_i \) is an i.i.d Laplace random variable drawn from the distribution \( \text{Lap}(\frac{\Delta}{2}) \), where \( b \) is the contribution upper bound for each record. Given \( \beta \in (0, 1) \), according to Corollary 11 we obtain that \( \Pr \left[ c_k - \tilde{c}_k \geq \alpha \right] \leq \beta \), such that \( \alpha = \frac{2 \Delta \sqrt{k \log \frac{1}{\beta}}}{b} \). Knowing that \( \tilde{c}_k - c_k \) computes the total number of records delayed after \( k \)-th updates, thus the theorem holds. Knowing that \( k' \leftarrow \frac{\Delta}{2} \sqrt{k \log \frac{1}{\beta}} \leq k T \), and according to Corollary 11, \( \sum_{i=1}^{k} Y_i \) is bounded by \( O(\frac{2b\sqrt{kT}}{e}) \), we conclude that the dummy data after \( k \)-th updates is bounded by \( O(\frac{2b\sqrt{kT}}{e}) + \frac{kbT}{e} \). \( \square \)

### B.3 Proof of Theorem 6

**Proof.** Let \( t \) denotes the current time, \( c_t \) counts how many records received since last update at every time \( t \). Assuming there \( k \) updates happened before current time \( t \), and thus we have \( k \) noisy thresholds \( \tilde{\theta}_1, \tilde{\theta}_2, \ldots, \tilde{\theta}_k \). Let \( A = \{a_1, a_2, \ldots, a_k\} \) as the collection of ant’s outputs, where \( a_i \in A \) is either \( \perp \) (no updates) or equals to \( c_j + \text{Lap}(\frac{\Delta b}{e}) \). According the Fact 3.7 in [31], such that

\[
\Pr \left[ \forall j, |\tilde{\theta}_j - \theta| \geq \alpha \right] = e^{-\frac{\alpha^2}{4}} \Rightarrow \Pr \left[ \sum_{j=1}^{k} |\tilde{\theta}_j - \theta| \geq \alpha \right] = k \times e^{-\frac{\alpha^2}{4}}
\]

(5)

Let \( k \times e^{-\frac{\alpha^2}{4}} \) to be at most \( \beta/4 \), then \( \alpha \geq \frac{16b\log(\log(2)/\beta)}{e} \). Similarly, for each time \( t \), we know that \( c_t - c_i = \text{Lap}(\frac{\Delta b}{e}) \), where \( c_i \) is the value used to compare with the noisy threshold, it satisfies:

\[
\Pr \left[ \forall 0 \leq i \leq t, |\tilde{c}_i - c_i| \geq \alpha \right] \leq e^{-\frac{\alpha^2}{4}} \Rightarrow \Pr \left[ \sum_{i=1}^{t} |\tilde{c}_i - c_i| \geq \frac{\alpha}{2} \right] \leq \sum_{i=1}^{t} (t - i - 1) \times e^{-\frac{\alpha^2}{4}} \leq t \times e^{-\frac{\alpha^2}{4}}
\]

(6)

where \( t \) denotes the time stamp for \( j \)-th update, and let \( t \times e^{-\frac{\alpha^2}{4}} \) be at most \( \beta/4 \), we have \( \alpha \geq \frac{16b\log(\log(2)/\beta)}{e} \). Finally, we set the following conditions \( \forall i : a_i \neq \perp, |a_i - c_i| = \text{Lap}(\frac{\Delta b}{e}) \) \( \geq \alpha \) holds with probability at most \( \beta/4 \), we obtain \( \alpha \geq \frac{16b\log(\log(2)/\beta)}{e} \). By combining the above analysis, we can obtain if set \( \alpha \geq \frac{16b\log(\log(2)/\beta)}{e} \) the following holds.

\[
\Pr \left[ \sum_{i=1}^{t} |\tilde{c}_i - c_i| \geq \frac{\alpha}{2} \right] \leq \beta.
\]

(7)

according to Eq. 7, with probability at most \( \beta \), the number of deferred data, \( \sum_{i=1}^{t} |c_i - a_i| \) is greater than \( \alpha \)

\[
\geq \frac{16b\log(\log(2)/\beta)}{e}.
\]

thus the total number of deferred data is bounded by \( O(\frac{16b\log t}{e}) \).

\( \square \)

### C SECURITY PROOF

In this section we continue to provide the complete formal security proof for IncShrink framework. We first provide privacy proofs for mechanisms \( M_{\text{timer}} \) and \( M_{\text{mem}} \) provided in Theorem 7 and 8.

**Theorem 12.** \( M_{\text{timer}} \) provided in Theorem 7 satisfies \( \epsilon \)-DP.

**Proof.** First we construct \( M_{\text{unit}}(X, \epsilon) \leftarrow f(X) + \text{Lap}(\frac{\Delta}{2}) \) where \( f = \sum_{x \in X} 1[x \neq \perp] \), and:

\[
\Delta f = \max_{\forall U_x, U_y \in \mathcal{X} \setminus \perp} |f(U_x) - f(U_y)| \leq \left| x \right| \leq e^{-\frac{\epsilon}{2}}
\]

Let \( X' \) denotes all possible inputs, and let \( U_x \in X' \), and \( U_y \in X \), denote two neighboring inputs (differ by one tuple). Next, let \( p_x \), \( p_y \) denote the density functions of \( M_{\text{unit}}(U_x, \epsilon) \), and \( M_{\text{unit}}(U_y, \epsilon) \), respectively. We compare the two terms under arbitrary point \( z \):

\[
p_x(z) = \frac{1}{2e} e^{-\frac{|f(U_x) - x|}{|f(U_x) - x|}} \leq e^{-\frac{|f(U_y) - x|}{|f(U_x) - x|}} \leq e^{-\frac{|f(U_y) - x|}{|f(U_x) - x|}} \leq e^{-\frac{|f(U_y) - x|}{|f(U_x) - x|}}
\]

(9)
Therefore, the privacy of the composed mechanism \( M_\text{timer}(D) \) is reduced to proving the privacy of the composed mechanism \( M = \{ M(\Lambda_{i-T}^T)^{(i+1)} D) \}_{i=1,2,3...} \).

Now consider two neighboring databases \( D \) and \( D' \) that differs in one logical update \( u_i \). Then we compute

\[
\ln \left( \frac{\Pr[M(D) = o_j]}{\Pr[M(D') = o_j]} \right) = \ln \left( \prod_{i=1}^{m} \frac{\Pr[M(\Lambda_{i-T}^T) = o_j]}{\Pr[M(\Lambda_{i-T}^T) = o_j]} \right)
\]

\[
= \ln \left( \prod_{i=1}^{m} \frac{\Pr[M(\Lambda_{i-T}^T) = o_j]}{\Pr[M(\Lambda_{i-T}^T) = o_j]} \right) \leq \epsilon
\]

Therefore, \( M_\text{timer} \) satisfies \( \epsilon \)-DP.

\( \square \)

**Theorem 13.** \( M_\text{ant} \) provided in Theorem 8 satisfies \( \epsilon \)-DP.

**Proof.** First we provide mechanism NANT (Numeric Above Noisy Threshold) in Algorithm 5, and proves its privacy guarantee.

**Algorithm 5 Numeric Above Noisy Threshold**

**Input:** data stream \( X \), privacy budget \( \epsilon \), threshold \( \theta \).

1. \( \epsilon_1 \leftarrow \frac{\epsilon}{2}, \epsilon_2 \leftarrow \frac{\epsilon}{2} \)
2. \( \tilde{\theta} \leftarrow \theta + \text{Lap}\left(\frac{\Delta_f}{\epsilon_2}\right) \), \( c \leftarrow 0 \)
3. for \( t \leftarrow 1, 2, \ldots \) do
4. \( u_t \leftarrow \text{Lap}\left(\frac{\Delta_f}{\epsilon_2}\right) \)
5. \( c \leftarrow f(X, t) \)
6. if \( c + u_t \geq \tilde{\theta} \) then
7. \( \text{output } \tilde{c} \leftarrow c + \text{Lap}\left(\frac{2\Delta_f}{\epsilon_2}\right) \), return
8. else
9. output 0

where \( f(X, t) \leftarrow \sum_{i=m}^{m} 1 \mid x_i \in X \wedge \) t \( \neq 0 \). We now prove it’s privacy as follows:

We start with a modified mechanism \( M_\text{unit} \) of NANT such that where \( i \) \ beyond \( i \) \( \frac{\Delta_f}{\epsilon_2} \) is satisfied (Alg 5:6), and outputs \( \bot \) for all other cases (Alg 5:9). We write the output of \( M_\text{unit} \) as \( O = \{ o_1, o_2, \ldots, o_m \} \), where \( V = \{ o_1, \ldots, o_m \} \), and \( O_m = \bot \). Now given two neighboring database \( X \) and \( X' \), and for all \( i, \) \( M \left[ \tilde{c}_i < x \right] \right) \left[ \tilde{c}_i < x + 1 \right] \) is satisfied, where \( \tilde{c}_i \) and \( \tilde{c}_i' \) denotes the \( \tilde{i}^{\text{th}} \) noisy count when applying \( M_\text{unit} \) over \( X \) and \( X' \)

respectively, such that:

\[
\Pr[M_\text{sparse}(U) = O]
\]

\[
= \int_{-\infty}^{\infty} \Pr[\tilde{\theta} < x]\left( \prod_{i=1}^{m} \Pr[\tilde{c}_i < x] \Pr[\tilde{c}_m \geq x] dx \right)
\]

\[
\leq \int_{-\infty}^{\infty} \epsilon_2^2 \Pr[\tilde{\theta} < x + 1]\left( \prod_{i=1}^{m} \Pr[\tilde{c}_i' < x + 1] \right)
\]

\[
\times \epsilon_2^2 \Pr[\tilde{c}_m \geq x + 1] \right) \Pr[\tilde{c}_m \geq x + 1] dx \right)
\]

\[
= \epsilon^2 \Pr[M_\text{sparse}(U) = O]
\]

Thus, \( M_\text{unit} \) satisfies \( \epsilon \)-DP. Moreover, mechanism NANT can be expressed as the composition of a \( M_\text{unit} \) and a Laplace mechanism, each with a privacy parameter of \( \frac{\epsilon}{2} \). Therefore by sequential composition, NANT satisfies \( \epsilon \)-DP. Similar, \( M_\text{ant} \) can be treated as repeatedly running NANT over disjoint data, thus by Eq. 11, \( M_\text{ant} \) satisfies \( \epsilon \)-DP.

\( \square \)

**Definition 5 (Secure 2-Party Computation [57]).** Let \( f = (f_1, f_2) \) be a functionality and let, \( \pi \) to be a 2 parrot protocol that computes \( f \). We say that \( \pi \) securely computes \( f \) in the presence of semi-honest adversaries if there exists p.p.t. simulator \( S_1 \) and \( S_2 \):

\[
\{ S_1(x, f_1(x, y), f_1(x, y), c(x, y) \} \subseteq \{ \text{VIEW}_\pi \_1(x, y), \text{output}_\pi \_1(x, y) \}
\]

\[
\{ S_2(x, f_2(x, y), f_2(x, y), c(x, y) \} \subseteq \{ \text{VIEW}_\pi \_2(x, y), \text{output}_\pi \_2(x, y) \}
\]

where \( c \) means computational indistinguishable, \( \text{VIEW}_\pi \) and \( \text{output}_\pi \) denotes the views and outputs when evaluating protocol \( \pi \).

**Theorem 14.** If there exists secure 2-PC protocols that satisfy Definition 5 and (2, 2)-secret sharing scheme that satisfy Lemma 9, then IncShrink implemented with sDPView and sDPANT view update protocol satisfies \( \epsilon \)-SIM-CDP.

**Proof.** In this section, we focus on proving the simulator provided in Table 1 yields computationally indistinguishable outputs compared to the execution of the real view update protocols. Let \( f_1(x, y) \) to be the functionality of truncated view transformation and \( f_2(x, y) \) to be the functionality of synchronizing data from secure cache to materialized view. \( \pi_1 \) and \( \pi_2 \) are the protocols that securely computes these two functionalities (Transform and Shrink). In general, we assume the secure cache and the materialized view are the secret-shared objects across the 2-PC participants.

Therefore, at each time \( t \), the adversary’s view against the entire view update protocol can be formulated as:

\[
\text{VIEW}_\pi^\_j(x, x_{1-j}, t) = \{ y^j_{f-1, i, j, f-1, i, j, c, j, \theta_i} \}
\]

where \( y^j_{f-1, i, j, f-1, i, j, c, j, \theta_i} \) and \( \theta_i \) denotes the corresponding secret shares party \( j \) obtains for user uploaded data, transformed view tuples, synchronized data, flushed data, cardinality counter and the noisy threshold (this parameter is not included in the view of sDPView protocol). Since \( y^j_{f-1, i, j, f-1, i, j, c, j, \theta_i} \) is the secret shared data generated
by owners, thus by Lemma 9,
\[ y_j^r = B \xrightarrow{rd} Z_m^r \text{ if } |y_j^r| = |B| \]
Since \( y_j^r \) is computed from the secure 2PC protocol \( \pi \), thus by Theorem 5, there must exist simulator such that \( S(x_j, f_j(x_j, x_{j-1})) \xrightarrow{\pi} y_j^r \). In addition, since \( y_j^r \) is assumed to be secret-shared data, thus
\[ \exists S, s.t. S(B \xrightarrow{rd} Z_m^r) \xrightarrow{\pi} y_j^r \text{ if } |y_j^r| = |B| \]
Similarly, we can also obtain
\[ \exists S, s.t. S(a, b \xrightarrow{rd} Z_m^r) \xrightarrow{\pi} c_j, \theta_j \]
Finally, since \( f_2 \) obliviously sorts then fetches from the recovered input data, thus the output of \( f_2 \) should be computational indistinguishable from the random sampling over the recovered input data. This also applies to cache flush, and therefore we can obtain
\[ \exists S, s.t. S(B, B' \xrightarrow{rd\text{-sample}} x) \xrightarrow{\pi} y_j^r, y'_j^r \text{ if } |B| = |y_j^r| \land |B'| = |y'_j^r| \]
As per the aforementioned analysis, the simulator provided in Table 1 yields computational indistinguishable transcripts in comparison with the real protocol execution. Since the simulator only takes in the outputs of differentially-private mechanisms and public parameters, thus the view update protocol satisfies \( \epsilon \)-SIM-CDP. □

D EXTENSION CONTINUED

D.1 Connecting with DP-Sync

We continue to provide more details regarding the utility guarantees when combining DP-Sync and IncShrink.

**Theorem 15 (Logical gap [83])**. For each time \( t \), the logical gap, \( LG_t \) between the outsourced and logical database is defined as the total number of records that have been received by the owner but have not been applied to the cache server.

In [83], logical gap is used as the major utility metric and typically a large logical gap indicates a relatively large error for queries to the outsourced database. Similar we derive a logical gap at time \( t \) for the materialized view as \( LG_t \), which denotes the number of view tuples delayed by the respective mechanisms (record synchronization strategy and view update protocol).

**Theorem 16 ((\( \alpha, \beta \))-accurate sync strategy).** A record synchronization strategy \( r\_sync \) over growing data \( D \) is \((\alpha, \beta)\)-accurate if there exists \( \alpha > 0 \), and \( 0 < \beta < 1 \), such that the logical gap when outsourcing \( D \) with \( r\_sync \) satisfies,

\[ \Pr[LG_t > \alpha] < \beta \]

**Theorem 17.** Applying IncShrink over the outsourced data uploaded by an \((\alpha, \beta)\)-accurate private synchronization strategy \( r\_sync \), results in error bounds \( O(ba_2 + \frac{2b}{\epsilon} \sqrt{2}) \) and \( O(ba_2 + \frac{6b \log T}{\epsilon}) \), respectively for sDPtimer and sDPantz protocol.

Proof. We provide the proof of the error bound under sDPtimer protocol and the bound under sDPantz can be proved using the same technique. Let \( \beta_1 \in (0, 1) \), and let \( \xi_k \) denotes the total number of cached view tuples that are delayed for synchronization after \( k^{th} \) view update. According to Theorem 4, \( \Pr[\xi_k > \frac{b \xi_k}{\epsilon}] < \beta_1 \), where \( b \) is contribution bound and \( \epsilon \) is privacy parameter. Let \( r\_sync \) is an \((\alpha_2, \beta_2)\)-accurate sync strategy, then the following holds

\[ \Pr[LG_t \geq \alpha] \leq \Pr[b \times LG_t + \xi_k \geq \alpha] \] (14)

By union bound, \( \Pr[b \times LG_t + \xi_k \geq \alpha_2 + \frac{2b}{\epsilon} \sqrt{2k \log \frac{1}{\beta_1}}] \leq \beta_1 + \beta_2 \), thus we obtain \( \Pr[LG_t \geq \alpha_2 + \frac{2b}{\epsilon} \sqrt{2k \log \frac{1}{\beta_1}}] \leq \beta_1 + \beta_2 \). This indicates the error is bounded by \( O(ba_2 + \frac{2b}{\epsilon} \sqrt{2k \log \frac{1}{\beta_1}}) \). The same proof technique can be used to prove the error bound of sDPantz. □

In general, as the logical gap of materialized view is resulted from the total data delayed by \( r\_sync \) and (ii) the total view entries delayed by Shrink protocol. Thus the logical gaps of the two mechanisms are additive.

D.2 Connecting with DP-Sync

To further define the effect of dummy records on overall computation cost, we introduce two efficiency metrics.

**Definition 6 (Filter Efficiency).** Given a Filter operator \( O \) with input \( O_1 \) of size \( n_1 \), let the number of dummy records in \( O_1 \) be \( Y_1(e_1) \), where \( e_1 \) is defined in privacy budget allocation \( P = e_1, \ldots, e_I \). The efficiency of \( O \) is defined as:

\[ E(P) = 1 - \frac{Y_1(e_1)}{n_1} \]

**Definition 7 (Join Efficiency).** Given a Join operator \( O \) whose inputs \( O_1 \) and \( O_2 \) are of size \( n_1 \) and \( n_2 \), respectively. Let the number of dummy records in \( O_1 \) and \( O_2 \) be \( Y_1(e_1) \) and \( Y_2(e_2) \), respectively, where \( e_1 \) and \( e_2 \) are defined in privacy budget allocation \( P = e_1, \ldots, e_I \). The efficiency of \( O \) is defined as:

\[ E(P) = 1 - \frac{Y_1(e_1) + Y_2(e_2)}{(n_1 + n_2)} \]

The total efficiency of a given query \( Q \) is defined as:

**Definition 8 (Query Efficiency).** Given a query \( Q \) comprised of operators \( O_1, O_2, \ldots, O_I \) with efficiencies \( E_1, \ldots, E_I \) and operator output cardinalities \( |O_i| \), respectively. The efficiency of \( Q \) with a privacy budget allocation \( P \) and total output size \( |O_{total}| \) is defined as:

\[ E_Q(P) = \frac{\sum_{i=1}^{I} |O_i|}{|O_{total}|} E_i(P) \]

Given a maximum privacy budget \( \epsilon \) and a maximum logical gap \( LG \), we can now define our optimization problem as follows:

\[ \max P E_Q(P) \text{ s.t. } \sum_{i=1}^{I} e_i = \epsilon, \sum_{i=1}^{I} LG_i \leq LG_{total}, e_i \geq 0 \forall i = 1, \ldots, I \] (15)

Note that in order to obtain the optimal privacy budget allocation, we require the true number of dummy records \( d_1 \) and \( d_2 \) in the inputs to each operator \( O \). However, revealing this information compromises our privacy guarantee. Instead, we can utilize estimates of \( d_1 \) and \( d_2 \) learned from the DP volume information released by our materialized joins, as seen in Figure 3.