PeGaSus: Data-Adaptive Differentially Private Stream Processing

Yan Chen
Duke University
Department of Computer Science
yanchen@cs.duke.edu

Ashwin Machanavajjhala
Duke University
Department of Computer Science
ashwin@cs.duke.edu

Michael Hay
Colgate University
Department of Computer Science
mhay@colgate.edu

Gerome Miklau
University of Massachusetts Amherst
College of Computing and Information Sciences
miklau@cs.umass.edu

ABSTRACT

Individuals are continually observed by an ever-increasing number of sensors that make up the Internet of Things. The resulting streams of data, which are analyzed in real time, can reveal sensitive personal information about individuals. Hence, there is an urgent need for stream processing solutions that can analyze these data in real time with provable guarantees of privacy and low error.

We present PeGaSus, a new algorithm for differentially private stream processing. Unlike prior work that has focused on answering individual queries over streams, our algorithm is the first that can simultaneously support a variety of stream processing tasks – counts, sliding windows, event monitoring – over multiple resolutions of the stream. PeGaSus uses a *Perturber* to release noisy counts, a data-adaptive *Perturber* to identify stable uniform regions in the stream, and a query specific *Smoother*, which combines the outputs of the *Perturber* and *Grouper* to answer queries with low error. In a comprehensive study using a WiFi access point dataset, we empirically show that PeGaSus can answer continuous queries with lower error than the previous state-of-the-art algorithms, even those specialized to particular query types.

1 INTRODUCTION

A number of emerging application domains rely on personal data processed in a streaming manner. Streaming data is the foundation of the Internet of Things [1] and prevalent in domains like environmental sensing, traffic management, health monitoring, and financial technology. Such data is typically captured and analyzed continuously and, because of the volume of the data, it is often processed as it arrives, in real time.

Since this data may report on individuals' location, health status, or other sensitive states, directly releasing the data, or even aggregates computed from the data stream, can violate privacy [8, 14]. In particular, continually updating statistics over time leaks more and more information to the attackers, potentially causing harmful privacy leakage [3].

Differential privacy [10], proposed over a decade ago, has become a primary standard for privacy. Informally, a (randomized) algorithm is differentially private if its output distribution is approximately the same when executed on two inputs that differ by the presence of a single individual's data. This condition prevents an attacker with access to the algorithm output from learning anything substantial about any one individual.

In this paper we propose a novel technique for releasing continuous query answers on real time streams under differential privacy. Our technique combines a *Perturber*, which generates a stream of noisy counts, and an independent module called a *Grouper*, which computes a partition of the data received so far. The *Grouper* privately finds partitions of the data which have small absolute deviations from their average. The final module, called the *Smoother*, combines the output of both the *Perturber* and the *Grouper*, generating the final private estimate of a query answer at each time step. The *Perturb-Group-Smooth* technique (we name it "PeGaSus") is data-adaptive: it offers improved accuracy for streams that have sparse or stable counts because the *Grouper* detects these regions and the *Smoother* uses knowledge of stability within these regions to infer better estimates.

PeGaSus not only helps release accurate differentially private streams (individual counts at each time step) but can also simultaneously support multiple alternative query workloads including sliding window queries and event monitoring queries like finding jumping and dropping points or detecting low signal points in the stream. These different tasks can be solved by reusing the output of the *Perturber* and the *Grouper*, and simply modifying the *Smoother* method, without incurring any additional privacy budget. Surprisingly, for many of these workloads, using our data dependent strategy outperforms state-of-the-art algorithms that are designed specifically for the corresponding query workload.

We propose extensions to PeGaSus to answer counting queries at different hierarchical resolutions on the stream. These extensions allow us to model typical query workloads that appear in building or network monitoring, where analysts are interested in both streams generated by individual sensors (or IP addresses), but also in aggregate streams generated by groups of sensors (or groups of IP addresses).

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134102

In summary, we make the following contributions:

- We design PeGaSus, a novel algorithm for answering a large class of continuous queries over real time data streams under differential privacy.
- PeGaSus uses a combination of a *Perturber*, a data-adaptive *Grouper* and a query specific *Smoother* to simultaneously support a range of query workloads over multiple resolutions over the stream.
- The *Grouper* and *Smoother*, in combination, offer improved accuracy for streams that have sparse or stable counts.
- A thorough empirical evaluation, on a real data stream collected from 4000 WiFi access points from a large educational institution, shows that by using different query specific *Smoother* methods, PeGaSus outperforms the previous state-of-the-art algorithms specialized to given workloads. For example, our data dependent algorithm can compute more accurate sliding window queries than the previous state-of-the-art algorithm that is designed for a specific sliding window workload.

The paper is organized as follows. Section 2 reviews the streaming data model, queries on streams, and the semantics of privacy on streams. In Section 3, we describe the framework of Perturb-Group-Smooth (PeGaSus) algorithm. In Section 4, we show how the framework can support multiple query workloads by applying different query specific *Smoother* methods. In Section 5, we discuss how to extend PeGaSus to answer counting queries at different hierarchical resolutions on the stream. Comprehensive experiments on a real data stream are presented in Section 6. Related work is discussed in Section 7 and our conclusions are in Section 8.

2 PRELIMINARIES

2.1 Stream data model

We define the source stream $\mathbf{D}$ as an infinite sequence of tuples. Each tuple is of the form (u, s, t) and is an element from the domain dom $= \mathcal{U} \times \mathcal{S} \times \mathcal{T}$ where $\mathcal{U}$ is set of user identifiers, $\mathcal{S}$ is a set of possible states, and $\mathcal{T}$ is an (infinite) set of timestamps. Each (u, s, t) records an atomic event, namely that user u was observed in state s at time t. Note that this single stream could contain events from multiple sources – these would be encoded as different states (elements of $\mathcal{S}$).

To simplify presentation, we represent time using logical timestamps letting $\mathcal{T} = \{1, 2, 3, \dots\}$. The rationale is that the analysis tasks we consider in Section 2.2 emit an aggregate summary of the stream periodically (e.g. every five minutes) and thus logical time $t = 1$ can be understood as capturing all events that happened within the first reporting period. Furthermore, this implies that a tuple (u, s, t) does not describe a specific, instantaneous event but rather it encodes the aggregate behavior of user u during the time step t. Therefore, the states $\mathcal{S}$ encode the state of a user during the logical time step t. We illustrate with an example.

Example 2.1. Consider a data stream management system that collects data from WiFi access points (APs) distributed across buildings on a campus. Users correspond to MAC addresses of individual devices that connect to WiFi access points. The set of time steps could represent the aggregated activity of a user over time intervals of 5 minutes each. Thus, time steps t and $t + 1$ would differ in wall clock time of 5 minutes. Finally, if there are m WiFi access points on campus, then we could have $m + 1$ states: a state $s_\perp$ that represents “user did not make a successful connection to any AP", and m states s_p, one for each AP p, that represents “user made at least one successful connection to the AP p".

The tuples in $\mathbf{D}$ arrive in order by time. Thus if (u', s', t') arrives after (u, s, t) it must be that $t \leq t'$. We use $\mathbf{D}_t$ to represent a stream prefix: the set of tuples that arrive on or before time t.

2.2 Queries on streams

We consider a number of queries on the private stream $\mathbf{D}$. The answer to a query on $\mathbf{D}$ is itself a stream. We focus on counting queries as well as other queries that can be derived from them.

2.2.1 Queries on a single target state. A counting query takes a specific target state s and reports, for each time step t, the number of users who were observed in state s at time t. More formally, let $C(s)$ be the infinite stream $C(s) = c_1(s), c_2(s), \dots$ where $c_t(s) = |\{(u', s', t') \in \mathbf{D}_t \mid t' = t \text{ and } s' = s\}|$. Let $C_t(s)$ denote the prefix stream of $C(s)$ up to time t. When clear from context, we drop the s and just use $C = c_1, c_2, \dots$.

Note that the answer to the query should be generated in “real time” – i.e., c_t should be produced before any tuple $(\cdot, \cdot, t + 1)$ is observed.

Example 2.2. An analyst might want to visualize the counts of the number of users who had at least one successful connection in a time step at access point AP_1. Hence, the target state is s_{AP_1}, $C_t(s_{AP_1})$ represents the number of users with at least one successful connection to AP_1 in time step t, and $C(s_{AP_1})$ represents the stream of counts.

The counting query defined above is referred to as a unit query. We can also support additional queries, all of which can be derived from C.

Sliding Windows. A sliding window query with window size w and target state s reports, for each time step t, the total number of times a user has been observed in state s in the most recent w time steps. More formally, let $SW(s, w)$ be an infinite stream $SW(s, w) = sw_1, sw_2, \dots$, where $sw_t(s, w) = |\{(u', s', t') \in \mathbf{D}_t \mid t - w < t' \leq t \text{ and } s' = s\}|$. Observe that the sliding window query answers can also be derived by summing corresponding counts in query $C(s)$: $sw_t(s, w) = \sum_{t'=t-w+1}^{t} c_{t'}(s)$.

Event Monitoring. While each tuple in the stream $\mathbf{D}$ captures an atomic event, the analyst may be interested in monitoring certain patterns in the event stream. We call this task event monitoring and consider monitoring event patterns that can be derived from the counting query stream C.

We define the event monitoring query as follows. Let $EM(s, w, f, B)$ $= b_1, b_2, \dots$ be an infinite stream of bits where $b_1 = 1$ if the monitored event has occurred at time t and 0 otherwise. The inputs to the query EM are the target state s, a desired window size w, an abitrary function f that computes on a window of w counts, and a boolean function B that computes on the ouptut of f. The bit b_t is computed as $b_t = B(f(c_{t-w+1}, \dots, c_t))$. We give two examples of event monitoring queries.

- Jumping and dropping point: This query monitors whether the count has changed by at least δ from the count w time steps ago. Thus, f computes the absolute difference between the current count and the count received w time steps before, $f(c_{t-w+1}, \dots, c_t) = |c_t - c_{t-w+1}|$ and B compares that difference to a threshold δ, $B(x) = 1$ if $x \geq \delta$ and is 0 otherwise.
- Low signal: This query monitors whether the total count in a sliding window is smaller than δ. Thus, f computes the total count, $f(c_{t-w+1}, \dots, c_t) = \sum_{t'=t-w+1}^{t} c_{t'}$ and B is again a threshold function, $B(x) = 1$ if $x < \delta$ and is 0 otherwise.

2.2.2 Queries on multiple target states. Our approach also supports queries on multiple target states. Let $\{s_1, \dots, s_m\} \subseteq \mathcal{S}$ denote the set of states the analyst is interested in. We support three variants. First, the analyst can simply issue multiple queries where each query is on a single target state (i.e., any one of the queries defined previously). We illustrate this with an example.

Example 2.3. An analyst might be interested in the unit query for the states s_p corresponding to all access points p within a specific building, as well as a low signal event monitoring query $EM(s_q, w, \dots)$, for all access points q housed in conference rooms across campus.

Second, we also support queries on *aggregations* of target states. We denote a single aggregation as $agg \subseteq \{s_1, s_2, \dots, s_m\}$. Any query that is defined for a single target state can also be defined over an aggregation of target states by replacing any equality condition on the state ($s' = s$) with set membership condition ($s' \in agg$). For example, an aggregated counting query is denoted $C(agg)$ and it produces a stream of answers $C(agg) = c(agg)_1, c(agg)_2, \dots$ where $c(agg)_t = \sum_{i \in agg} c_t(s_i)$.

Finally, the analyst may wish to ask a query about more than one aggregation of states. Let $AGG = \{agg_1, agg_2, ...\}$ denote a collection of aggregations. We consider the special case where this collection has a hierarchical structure.

Definition 2.4. A set of aggregations $AGG = \{agg_1, agg_2, ...\}$ is *hierarchical* if for any two aggregations $agg_1, agg_2 \in AGG$, they satisfy one of the following two properties:

(1) $agg_1 \subset agg_2$ or $agg_2 \subset agg_1$.
(2) $agg_1 \cap agg_2 = \emptyset$.

Intuitively, a set of hierarchical aggregations AGG can be represented as a tree or a forest, where any child aggregation contains a subset of states that its parent aggregation covers, and any two child aggregations with the same parent aggregation cover two disjoint sets of states. We use $level(agg \mid AGG)$ to denote the level of $agg \in AGG$. $level(agg \mid AGG) = 1$ if agg has no parent aggregation in AGG. If $agg_1, agg_2 \in AGG$ and agg_1 is the child aggregation of agg_2, $level(agg_1 \mid AGG) = level(agg_2 \mid AGG) + 1$.

Example 2.5. An analyst might be interested in aggregate counts of successful connections at the level of individual sensors, as well as for aggregations of sensors within the same room, same floor, and the same building on campus. For each room r, let agg_r denote the aggregation of states s_p, where p is an access point in room r. Similarly, for a floor f and building b, we can define aggregations agg_f and agg_b that aggregate states s_p, where p is in the floor f or building b, respectively. These sets of aggregations form a hierarchy. An analyst might be interested in the unit query for aggregate states at floors and buildings as well as a low signal event monitoring query $EM(agg_r, w, \dots)$, for each rooms r.

2.3 Privacy for Streams

Two stream prefixes $\mathbf{D}_t$ and $\mathbf{D}'_t$ are considered *neighbors* if they differ by the addition or removal of a single tuple; i.e., $|\mathbf{D}_t \oplus \mathbf{D}'_t| = 1$ where $\oplus$ indicates symmetric difference. The algorithms we consider in this paper operate on stream prefixes.

We define privacy for streams as follows, analogous to event differential privacy [2, 11].

Definition 2.6 (ϵ-differential privacy). Let $\mathcal{A}$ be a randomized algorithm that takes as input a stream prefix of arbitrary size and outputs an element from a set of possible output sequences $\mathcal{O}$. Then $\mathcal{A}$ satisfies ϵ-differential privacy if for any pair of neighboring stream prefixes $\mathbf{D}_t$ and $\mathbf{D}'_t$, for all t, and $\forall O \subseteq \mathcal{O}$,

$$Pr[\mathcal{A}(\mathbf{D}_t) \in O] \leq e^{\epsilon} \times Pr[\mathcal{A}(\mathbf{D}'_t) \in O]$$

The parameter ϵ controls the privacy risk and smaller ϵ corresponds to stronger privacy protection. The semantics of this privacy guarantee are discussed further in Section 2.4.

The following composition properties hold for differentially private algorithms, each commonly used for building complex differentially private algorithms from simpler subroutines. Suppose $\mathcal{A}_1(\cdot)$ and $\mathcal{A}_2(\cdot)$ are ϵ_1- and ϵ_2-differentially private algorithms, respectively.

- *Sequential Composition:* Computing $\mathcal{A}_1(\mathbf{D}_t)$ and $\mathcal{A}_2(\mathbf{D}_t)$ satisfies $(\epsilon_1 + \epsilon_2)$-differential privacy for any $\mathbf{D}$.
- *Parallel Composition:* Let A and B be disjoint subsets of dom. Computing $\mathcal{A}_1(\mathbf{D}_t \cap A)$ and $\mathcal{A}_1(\mathbf{D}_t \cap B)$, satisfies ϵ_1-differential privacy.
- *Postprocessing*: For any algorithm $\mathcal{A}_3(\cdot)$, releasing $\mathcal{A}_3(\mathcal{A}_1(\mathbf{D}_t))$ still satisfies ϵ_1-differential privacy for any $\mathbf{D}$. That is, postprocessing an output of a differentially private algorithm does not incur any additional loss of privacy.

The composition properties allow us to execute multiple differentially private computations and reason about the cumulative privacy risk. In our applications, we want to bound the total risk so we impose a total epsilon "privacy budget" and allocate a portion of the budget to each private computation.

An arbitrary numerical function f can be made differentially private by adding noise to its output. The amount of noise depends on the *sensitivity* of the function.

Definition 2.7 (Sensitivity). Let f be a function that maps datasets to $\mathbb{R}^n$. The *sensitivity* denoted as $\Delta(f)$, is defined to be the maximum L_1 distance between function outputs from any two neighboring data streams $\mathbf{D}_t$ and $\mathbf{D}'_t$.

$$\Delta(f) = \max_{\mathbf{D}_t, \mathbf{D}'_t : |\mathbf{D}_t \oplus \mathbf{D}'_t| = 1} ||f(\mathbf{D}_t) - f(\mathbf{D}'_t)||_1.$$

The *Laplace Mechanism* [10] achieves differential privacy by adding noise from Laplace distribution calibrated to the sensitivity.

Definition 2.8 (Laplace Mechanism (LM)). Given a function f that maps datasets to $\mathbb{R}^n$, the Laplace Mechanism outputs $f(\mathbf{D}_t) + \eta$, where η is a vector of independent random variables drawn from a Laplace distribution with the probability density function $p(x|\lambda) = \frac{1}{2\lambda} e^{-|x|/\lambda}$, where $\lambda = \Delta(f)/\epsilon$.

Remark. We make an assumption that each user can be in at most m different states s during one time period. Without loss of generality, we assume $m = 1$ in our application. If $m > 1$, we can simply normalize the counts (a user who is in m states at time t contributes $1/m$ rather than 1 to the corresponding counting queries) or increase the amount of noise injected in our algorithm in order to provide privacy protection in terms of any single user during one time period.

2.4 Privacy Semantics

In this section, we discuss the semantics of privacy ensured by Definition 2.6 and justify our choice of this privacy goal.

The privacy ensured by Definition 2.6 can be interpreted in terms of (a) plausible deniability, and (b) disclosure of secrets to adversaries. Let $\phi_u(t)$ and $\phi'_u(t)$ be two mutually exclusive boolean properties about a user u at time step t. Examples of such properties could be that a user was in building B_1 at time t and building B_2 at time t, respectively. An algorithm M satisfying Definition 2.6 allows a user to deny that $\phi'_u(t)$ is true rather than $\phi_u(t)$ since for all neighboring streams $\mathbf{D}_t, \mathbf{D}'_t$ such that $\phi_u(t)$ is true on $\mathbf{D}_t$ and $\phi'_u(t)$ is true on $\mathbf{D}'_t$, and for all output sets $O \in range(\mathcal{A})$, we have:

$$Pr[\mathcal{A}(\mathbf{D}_t) = O] \leq e^{\epsilon} Pr[\mathcal{A}(\mathbf{D}'_t) = O]$$

Plausible deniability holds even for properties that span larger time windows, albeit to a lesser extent and degrades with the length of the time window. That is, if $\phi_u(t,k)$ and $\phi'_u(t,k)$ are two mutually exclusive boolean properties about a user u that span a time window of $[t-k+1, t]$, then for all streams $\mathbf{D}_t, \mathbf{D}'_t$ such that $\phi_u(t,k)$ is true on $\mathbf{D}_t$ and $\phi'_u(t,k)$ is true on $\mathbf{D}'_t$, and that differ only in the states in the time window $[t-k+1, t]$, and for all output sets $O \in range(\mathcal{A})$, we have:

$$Pr[\mathcal{A}(\mathbf{D}_t) = O] \leq e^{k \cdot \epsilon} Pr[\mathcal{A}(\mathbf{D}'_t) = O]$$

Thus our definition also captures the more general w-event privacy [16]. And, if a time step corresponds to 5 minutes, and an algorithm $\mathcal{A}$ satisfies Definition 2.6 with $\epsilon = 0.1$, then for properties that span 10 minutes, we get privacy at a level $\epsilon = 0.2$, and for properties that span 1 hour, we get privacy at a level of $\epsilon = 1.2$. If the protected window size goes to infinity (k is unbounded), one can still guarantee privacy with parameter $\ell \cdot \epsilon$, where ℓ is the maximum number of tuples in the stream corresponding to a single user. If both k and ℓ are unbounded, one can extend existing negative results [8] to show that it is impossible to release accurate statistics at each time and offer privacy.

Next we explore the semantics of Definition 2.6 in terms of disclosure of secrets to adversaries, which can be done in terms of the Pufferfish framework [18]. One can show that if an algorithm $\mathcal{A}$ satisfies Definition 2.6, then the adversary's posterior odds that $\phi_u(t)$ is true vs $\phi'_u(t)$ is true after seeing the output of $\mathcal{A}$, for any pair of mutually exclusive secrets that span a single time step, is no larger than e^{ϵ} times the adversary's prior odds. However, this strong privacy guarantee only holds under the restrictive assumptions that an adversary is not aware of possible correlations between a user's states across time steps. With knowledge of correlations, an adversary can learn sensitive properties of a user within a time step even from outputs of differentially private algorithms [17, 20, 24]. Nevertheless, the ratio of the adversary's posterior to prior odds is still guaranteed to be no larger than $e^{\ell\epsilon}$ even in the presence of correlations. Recall that ℓ is the maximum number of tuples in the stream corresponding to a single user.

Recent work [5, 22] has provided methods for deriving an $\epsilon' > \epsilon$ (but no more than $\ell \times \epsilon$), such that algorithms satisfying Definition 2.6 with parameter ϵ offer a weaker ϵ' bound on privacy loss even when records are correlated across time steps. For specific types of correlations, the effective privacy guarantee is closer to ϵ and much smaller than $\ell \times \epsilon$.

We emphasize that our algorithms are designed to satisfy Definition 2.6 with parameter ϵ, but simultaneously satisfy all of the above provable privacy guarantees, with a possibly different privacy parameter. Therefore, for the remainder of the paper, we focus exclusively on developing algorithms that satisfy Definition 2.6 while minimizing error.

3 PEGASUS STREAM RELEASE

In this section, we describe a novel, data-dependent method (called PeGaSus) for private, real-time release of query answers on data streams. Our algorithm consists of a novel combination of data perturbation and online partitioning, followed by post-processing.

We first present the algorithm for computing a unit counting query in a single target state. In Section 4, we explain how the algorithm can be adapted to answer other kinds of queries on a single target state and in Section 5, we explain an extension of the algorithm to support multiple queries over hierarchical aggregations of states. The input to the algorithm consists of the true answers to the unit counting query C. The output of the algorithm is $\hat{C} = \hat{c}_1, \hat{c}_2, \ldots$, an infinite stream where $\hat{c}_t$ is an estimate of the true answer c_t.

The three main modules of our method PeGaSus are as follows:

- ***Perturber*:** The *Perturber* consumes the input stream, adds noise to each incoming element of the stream, and releases a stream of noisy counts.
- ***Grouper*:** The *Grouper* consumes the input stream and groups elements of the stream seen so far.
- ***Smoother*:** The *Smoother* performs post-processing using the output of both above modules.

The *Perturber* is a standard noise-addition mechanism, but the *Grouper* carries out an important role of partitioning the data into regions that can be well-approximated by a uniform sub-stream. The *Smoother* then combines this information with the output of the *Perturber*. The result is a data-dependent algorithm which can reduce error for streams with properties that are commonly witnessed in practice.

The combination of the above three modules is formalized in Algorithm 1, called Perturb-Group-Smooth stream release (PeGaSus). When a new count c_t arrives at time t, the *Perturber* takes

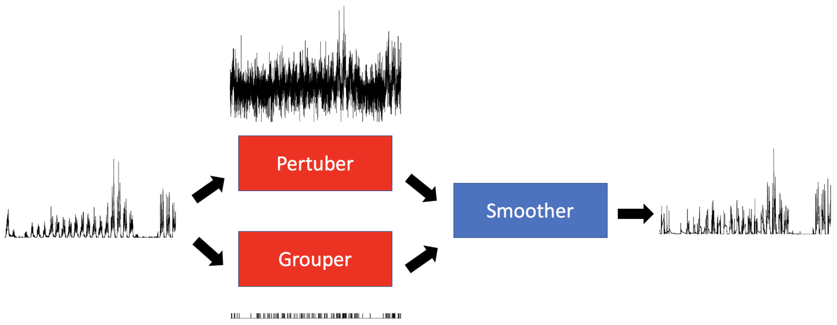


Figure 1: The PeGaSus algorithm for generating private streams. The *Perturber* and *Grouper* consume the true input stream, while the *Smoother* consumes their output and produces the final result stream.

Algorithm 1 Perturb-Group-Smooth based Stream Release (PeGaSus)

Input: $C = c_1, c_2, \ldots,$ privacy budget $\epsilon = \epsilon_p + \epsilon_g$
Output: Private stream $\hat{C} = \hat{c}_1, \hat{c}_2, \ldots,$

```
1: for each time $t$ do
2:     $\tilde{c}_t \leftarrow Perturber(c_t, \epsilon_p)$
3:     $P_t \leftarrow Grouper(C_t, P_{t-1}, \epsilon_g)$
4:     $\hat{c}_t \leftarrow Smoother(\tilde{C}_t, P_t)$
5:     Release $\hat{c}_t$
6: end for
```

the input c_t and outputs a noisy version $\tilde{c}_t$, using ϵ_p portion of the overall ϵ privacy budget (line 2). The *Grouper* takes as input all of the data received so far $C_t = c_1, \ldots, c_t$ and the partition from the previous time P_{t-1}. (Partition P_{t-1} is a partition over the integers $\{1, \ldots, t-1\}$ and represents a grouping of the first $t-1$ counts.) At each time step, the *Grouper* outputs an updated partition P_t with a portion of the privacy budget ϵ_g (in line 3). The *Smoother* then computes a final estimate $\hat{c}_t$ of c_t based on all the initial noisy counts $\tilde{C}_t = \tilde{c}_1, \ldots, \tilde{c}_t$ and the current partition P_t (in line 4).

The execution of Algorithm 1 is illustrated in Figure 1. Both the *Perturber* and the *Grouper* are colored red because they consume the input stream and use the privacy budget. The *Smoother* is colored blue because it only uses the output of the *Perturber* and the *Grouper*.

Theorem 3.1. *When the* Perturber *satisfies ϵ_p-differential privacy and the* Grouper *satisfies ϵ_g-differential privacy, Algorithm 1 ensures $\epsilon_p + \epsilon_g = \epsilon$-differential privacy.*

The above theorem follows directly from the sequential composition and post-processing properties of differential privacy (as described in Section 2).

Algorithm 1 forms the basis of a number of algorithm variants we consider throughout the paper. In the remainder of this section, we describe below the design of each module, and basic variants for the *Smoother*. We also include theoretical analysis that illustrates cases where smoothing can reduce error. Sections 4 and 5 describe extensions to the algorithm for other kinds of queries.

Algorithm 2 Deviation based Grouper (DBG)

Input: $C_t = c_1, \ldots, c_t$, the previous partition P_{t-1}, ϵ_g, θ
Output: P_t (a partition of $\{1, \ldots, t\}$)

```
1: if $t = 1$ then
2:     $P_{t-1} \leftarrow \emptyset$ and let $G$ be closed, empty group.
3: else
4:     $G \leftarrow$ Last group from $P_{t-1}$
5: end if
6: if $G$ has been closed then
7:     $P_t \leftarrow P_{t-1} \cup \{\{t\}\}$ and let the last group $\{t\}$ be open.
8:     $\tilde{\theta} \leftarrow \theta + Lap(4/\epsilon_g)$
9: else
10:    $\tilde{\theta} \leftarrow \tilde{\theta}_{prev}$        ▷ $\tilde{\theta}_{prev}$ cached from previous call.
11:    if $(dev(C_t[G \cup \{t\}]) + Lap(8/\epsilon_g)) < \tilde{\theta}$ then
12:        $P_t \leftarrow P_{t-1}$ with $G$ replaced by $G \cup \{t\}$ and $G$ still open.
13:    else
14:        $P_t \leftarrow P_{t-1} \cup \{\{t\}\}$ and close both group $G$ and $\{t\}$.
15:    end if
16: end if
17: $\tilde{\theta}_{prev} \leftarrow \tilde{\theta}$        ▷ cache $\tilde{\theta}_{prev}$ for subsequent calls.
18: Return $P_t$
```

3.1 Design of the *Perturber*

The *Perturber* takes input c_t at each time t and outputs a noisy version $\tilde{c}_t$. We use the Laplace Mechanism (Section 2.3) as the implementation of the *Perturber*.

3.2 Design of the *Grouper*

Next we describe the Deviation-based Grouper (DBG) (Algorithm 2), a differentially private method for online partitioning which chooses partitions that approximately (subject to distortion introduced for privacy) minimizes a quality score based on deviation.

Recall that this module runs independently of the *Perturber* and does not consume its output. Instead the *Grouper* takes as input all of the data received so far, $C_t = c_1, \ldots, c_t$ at time t, and outputs a partition of C_t. At any time t, the stream seen so far has been partitioned into contiguous groups. All but the most recent group are *closed* and will not be changed, but the most recent group may continue to grow as new elements arrive, until it is eventually closed.

To evaluate the quality of a potential group, we use the deviation function, which measures the absolute difference of a set of counts from its average. Let G be a group of indexes of data $C_t = c_1, \ldots, c_t$, $C_t[G]$ be the set of corresponding counts in C_t and denote by $|G|$ the size of the group. Then the deviation of $C_t[G]$ is denoted $dev(C_t[G])$ and is defined:

$$dev(C_t[G]) = \sum_{i \in G} \left| c_i - \frac{\sum_{i \in G} c_i}{|G|} \right|.$$

When the $dev(C_t[G])$ is small, the counts in $C_t[G]$ are approximately uniform and the *Smoother* can exploit this.

Algorithm 2 takes as input the stream seen so far, C_t, the latest partition, P_{t-1}, along with ϵ_g and a user-defined deviation threshold θ which influences the acceptable deviation in a partition. Because the algorithm uses the Sparse Vector Technique [13], it maintains a noisy threshold $\tilde{\theta}_{prev}$ used at the previous time. When a new data c_t arrives at time t, we check the status of the last group G from the previous partition P_{t-1}. If G is closed, we put c_t into a new open group and reset the noisy threshold (in lines 6-8). Otherwise, we compute the deviation of the $C_t[G \cup \{t\}]$ and compare a noisy version of the deviation value with the noisy threshold. If the noisy deviation is smaller than the noisy threshold, we add $\{t\}$ into the open G (in lines 11-12). Otherwise, we add a new group $\{t\}$ into the partition and close both G and $\{t\}$ (in line 14).

The following example explains how the *Grouper* defined in Algorithm 2 would run, assuming for simplicity an infinite privacy budget (so that the noise added in lines 8 and 11 is zero).

Example 3.2. Consider a stream of counts $C_5 = [5, 5, 6, 9, 10]$ and a threshold $\theta = 2$. At time 1, the *Grouper* generates a partition P_1 containing a single open group G with a single index: $G = \{1\}$ and $P_1 = \{G\}$. At time 2, the last group $G = \{1\}$ is open and $dev(C_2[\{1, 2\}]) = dev([5, 5]) = 0$. Because the deviation is less than θ, we add the current data into G and keep G open. P_2 still contains a single group G. At time 3, the last group $G = \{1, 2\}$ is open and $dev([5, 5, 6]) = \frac{4}{3} < \theta$. We still add the current data into G and keep G open. At time 4, the last group $G = \{1, 2, 3\}$ is open but $dev([5, 5, 6, 9]) = 5.5 > \theta$. Thus we close $G = \{1, 2, 3\}$ and create the second group $\{4\}$ which is also closed. Thus, $P_4 = \{\{1, 2, 3\}, \{4\}\}$. At time 5, the last group $G = \{4\}$ is closed, so we start with a new open group $G = \{5\}$. The final output at time 5 is $P_5 = \{\{1, 2, 3\}, \{4\}, \{5\}\}$.

With a realistic (non-infinite) setting for ϵ_g, the *Grouper* runs in a similar manner but noise is added to both the threshold and to the deviations that are compared with the threshold. Therefore, the output partitions will differ.

To prove the privacy of Algorithm 2 we will use the following lemma:

LEMMA 3.3 (SENSITIVITY OF DEVIATION [19]). *The global sensitivity of the deviation function $\Delta(dev)$ is bounded by 2.*

THEOREM 3.4. *Using Algorithm 2 to generate a partition at every time ensures ϵ_g-differential privacy.*

PROOF. The algorithm is an implementation of the *Sparse Vector Technique* [13] on multiple disjoint sub-streams applied to the dev function. The noise was injected to both the threshold and the deviation value in order to ensure privacy. Because the sensitivity of dev is bounded by 2 (Lemma 3.3), the noise added to the threshold, in line (4), is computed as $Lap(4/\epsilon_g) = Lap(2\Delta(dev)/\epsilon_g)$. In line (7), the noise added to the deviation value is $Lap(8/\epsilon_g) = Lap(4\Delta(dev)/\epsilon_g))$. Based on the original proof of Sparse Vector Technique from [13], this amount of noise ensures ϵ_g-differential privacy for each generated group. In addition, the derived streams from two neighboring source streams can only differ by 1 at one time. Thus, there will be only one generated group different in term of two neighboring source streams. By parallel composition, Algorithm 2 satisfies ϵ_g-differential privacy. □

3.3 Design of the *Smoother*

The *Smoother* computes the final estimate $\hat{c}_t$ for each c_t received at time t based on all the noisy counts $\tilde{C}_t$ from the *Perturber*, in combination with the current partition P_t. Suppose the last group $G = \{t - k, \ldots, t - 1, t\}$ from P_t contains current index t with the previous k indexes. Given this output from the *Grouper*, there are several post-processing methods we may apply to generate an estimate, $\hat{c}_t$, that improves upon $\tilde{c}_t$. These are alternatives for the *Smoother* in Algorithm 1:

1. *AverageSmoother*: We use the average noisy counts of the data indexed in group G to be the estimate of the current data.
$$\hat{c}_t = \frac{\sum_{i \in G} \tilde{c}_i}{|G|}.$$
2. *MedianSmoother*: We use the median noisy counts of the data indexed in group G to be the estimate of the current data.
$$\hat{c}_t = median\{\tilde{c}_i \mid i \in G\}.$$
3. *JSSmoother*: We apply the James-Stein estimator [23] to update the noisy count of the current data based on the noisy counts of the data indexed in group G.
$$\hat{c}_t = \frac{\tilde{c}_t - avg}{|G|} + avg,$$
where $avg = \frac{\sum_{i \in G} \tilde{c}_i}{|G|}$. We assume uniformity on each group and apply the James-Stein estimator to let each estimate shrink to the mean. (We can only use the noisy mean here in terms of the privacy.)

We theoretically analyze the effect of *AverageSmoother* in the next section and empirically evaluate all three variants in Section 6. We conclude this section with an example.

Example 3.5. Continuing from Example 3.2, we have a stream of true counts $C_5 = \{5, 5, 6, 9, 10\}$ with noisy counts from the *Perturber* of $\tilde{C}_5 = \{5.6, 4.4, 6.7, 9.5, 10.2\}$ and a final partition $P_5 = \{\{1, 2, 3\}, \{4\}, \{5\}\}$ from the *Grouper*. We now illustrate how the final estimates $\hat{C}_5$ would have been produced using *MedianSmoother* as *Smoother*. Recall that each $\hat{c}_t$ is released in real-time based on P_t, the groups at time t, and not the final grouping P_5. At time 1, $P_1 = \{\{1\}\}$ and the last group is $G = \{1\}$, $\hat{c}_1 = median\{5.6\} = 5.6$. At time 2, the last group is $G = \{1, 2\}$, $\hat{c}_2 = median\{5.6, 4.4\} = 5$. At time 3, the last group is $G = \{1, 2, 3\}$, $\hat{c}_3 = median\{5.6, 4.4, 6.7\} = 5.6$. At time 4, $P_4 = \{\{1, 2, 3\}, \{4\}\}$ and the last group is $G = \{4\}$, so $\hat{c}_4 = median\{9.5\} = 9.5$. At time 5, the last group is $G = \{5\}$,

$\hat{c}_5 = median\{10.2\} = 10.2$. Thus, the final estimates are $\hat{C}_5 = \{5.6, 5, 5.6, 9.5, 10.2\}$.

3.4 Error analysis of smoothing

We now formally analyze how the online grouping and post-processing *Smoother* may help for improving the accuracy of the output.

THEOREM 3.6. *Suppose one group G in the resulting partition from* Grouper *contains n indexes, $i+1, i+2, \ldots, i+n$. Assume that $\hat{C}$ is produced using the* AverageSmoother *as the* Smoother. *Then $\hat{C}[G]$, the resulting estimate for group G, will have lower expected error than $\tilde{C}[G]$, formally stated as*

$$\mathbb{E}\left\|\hat{C}[G]-C[G]\right\|_2 \le \mathbb{E}\left\|\tilde{C}[G]-C[G]\right\|_2$$

provided that the deviation of $C[G]$ satisfies

$$dev(C[G]) \le \frac{\sqrt{2(n-\ln n-1)}}{(1+\ln(n-1))\epsilon_p}$$

PROOF. In terms of $\tilde{C}$ which are generated from the *Perturber* by applying *Laplace Mechanism*, we have $\mathbb{E}\left\|\tilde{C}[G]-C[G]\right\|_2 = n \times \frac{2}{\epsilon_p^2}$. At each time $i{+}k$, $G = \{i{+}1, \ldots, i{+}k\}$. By using the *AverageSmoother*, $\hat{c}_{i+k} = \frac{\tilde{c}_{i+1}+\cdots+\tilde{c}_{i+k}}{k}$. Then the following holds:

$$\begin{aligned}
\mathbb{E}\left\|\hat{C}[G]-C[G]\right\|_2 &= \mathbb{E}\Big(\sum_{k=1}^{n}(\frac{\tilde{c}_{i+1}+\cdots+\tilde{c}_{i+k}}{k}-c_{i+k})^2\Big)\\
&= \sum_{k=1}^{n}E[(\frac{\tilde{c}_{i+1}+\cdots+\tilde{c}_{i+k}}{k}-c_{i+k})^2]\\
&= \sum_{k=1}^{n}E[(\frac{c_{i+1}+n_{i+1}+\cdots+c_{i+k}+n_{i+k}}{k}-c_{i+k})^2]\\
&= \sum_{k=1}^{n}E[(\frac{c_{i+1}+\cdots+c_{i+k}}{k}-c_{i+k}+\frac{n_{i+1}+\cdots+n_{i+k}}{k})^2]\\
&= \sum_{k=1}^{n}((avg_k-c_{i+k})^2+E[(\frac{n_{i+1}+\cdots+n_{i+k}}{k})^2]\\
&\quad +2*E[(\frac{c_{i+1}+\cdots+c_{i+k}}{k}-c_{i+k})*\frac{n_{i+1}+\cdots+n_{i+k}}{k}]).
\end{aligned}$$

Since $n_{i+1}, \ldots, n_{i+n}$ are independent Laplace noise with parameter $\frac{1}{\epsilon_p}$, $E[(\frac{n_{i+1}+\cdots+n_{i+k}}{k})^2] = \frac{1}{k}\times\frac{2}{\epsilon_p^2}$ and $E[\frac{n_{i+1}+\cdots+n_{i+k}}{k}] = 0$. Then we have

$$\begin{aligned}
&\mathbb{E}\left\|\hat{C}[G]-C[G]\right\|_2\\
=\ & \sum_{k=2}^{n}(avg_k-c_{i+k})^2+\sum_{k=1}^{n}\frac{1}{k}\times\frac{2}{\epsilon_p^2}\\
<\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+\sum_{k=2}^{n}(avg_k-c_{i+k})^2\\
\le\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+\sum_{k=2}^{n}(|avg_k-avg_n|+|avg_n-c_{i+k}|)^2\\
\le\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+(\sum_{k=2}^{n}|avg_k-avg_n|+\sum_{k=2}^{n}|avg_n-c_{i+k}|)^2\\
\le\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+(\sum_{k=2}^{n}|avg_k-avg_n|+dev(C[G]))^2.
\end{aligned}$$

Also, we have

$$\begin{aligned}
&|avg_k-avg_n| = |\frac{c_{i+1}+\cdots+c_{i+k}}{k}-avg_n|\\
=\ & |\frac{c_{i+1}-avg_n}{k}+\cdots+\frac{c_{i+k}-avg_n}{k}|\\
\le\ & \frac{1}{k}(|c_{i+1}-avg_n|+\cdots+|c_{i+k}-avg_n|)\\
\le\ & \frac{1}{k}(|c_{i+1}-avg_n|+\cdots+|c_{i+n}-avg_n|) = \frac{1}{k}dev(C[G]).
\end{aligned}$$

Thus,

$$\begin{aligned}
&\mathbb{E}\left\|\hat{C}[G]-C[G]\right\|_2\\
\le\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+(\sum_{k=2}^{n-1}\frac{1}{k}\times dev(C[G])+dev(C[G]))^2\\
<\ & (\ln n+1)\times\frac{2}{\epsilon_p^2}+(1+\ln(n-1))^2 dev(C[G])^2,
\end{aligned}$$

where $avg_k = \frac{c_{i+1}+\cdots+c_{i+k}}{k}$ for $k \in [1, n]$.

Therefore, when $dev(C[G]) \le \frac{\sqrt{2(n-\ln n-1)}}{(1+\ln(n-1))\epsilon_p}$, we have

$$\mathbb{E}\left\|\hat{C}[G]-C[G]\right\|_2 \le \mathbb{E}\left\|\tilde{C}[G]-C[G]\right\|_2$$

□

Theorem 3.6 implies that when the *Grouper* finds a group with large size but small deviation value, using the *AverageSmoother* can reduce the error of the estimates. Many realistic data streams are either sparse or have stable counts, which suggests that smoothing can help reduce error. Although we only theoretically analyze the *AverageSmoother* as the *Smoother*, we empirically compare the three different post-processing strategies on many real streams (in Section 6). We find that the *Grouper* often finds large groups with low deviation and *MedianSmoother* consistently generates the most accurate noisy streams under all settings. Thus, in our algorithms, we use the *MedianSmoother* as the default *Smoother* to compute the final estimates.

Algorithm 3 *Window Sum Smoother* (WSS)

```
Input: $\tilde{C}_t = \tilde{c}_1, \ldots, \tilde{c}_t, P_t, w$
Output: $\hat{sw}_t$
1: $\hat{sw}_t \leftarrow 0$
2: for each $G \in P_t$ such that $G \cap \{t - w + 1, \ldots, t\} \neq \emptyset$ do
3:     $\hat{c} \leftarrow median\{\tilde{c_i} \mid i \in G\}$
4:     $\hat{sw}_t \leftarrow \hat{sw}_t + \hat{c} \times |G \cap \{t - w + 1, \ldots, t\}|$
5: end for
6: Return $\hat{sw}_t$
```

4 SUPPORT FOR OTHER QUERIES

The previous section describes how the PeGaSus algorithm (Algorithm 1) can be used to generate $\hat{C}$, a differentially private answer to a given unit counting query C. In this section, we describe how to adapt the algorithm to answer the other queries described in Section 2.2, specifically sliding window queries and event monitoring queries.

A distinctive feature of our approach is that we use the same basic PeGaSus algorithm for these queries and change only the *Smoother*. This is possible because the answers to these queries can be derived from C. Our approach uses the noisy counts $\tilde{C}_t$ along with the group information P_t to do appropriate smoothing for the specific query. Recall that the *Smoother* does not take private data as input and only post-processes the outputs of the differentially private *Perturber* and *Grouper* subroutines. Therefore, we can swap out *Smoother* without impacting the privacy guarantee. An added benefit of this approach is that we can *simultaneously* support all three kinds of queries – counting, sliding window, and event monitors – all using a single privacy budget.

An effective *Smoother* should be designed in terms of the specific applications as well as users' knowledge about the input stream. For sliding window queries, we propose a new post-processing strategy called *Window Sum Smoother* (WSS), which is shown in Algorithm 3. At time t, for every $c_{t'}$ contained in the sliding window ($t' \in [t - w + 1, t]$), we use the *MedianSmoother* to compute an estimate $\hat{c}_{t'}$ for the count at time t'. To compute the answer to the sliding window query, we simply sum up the counts within the window w.

Note that this is subtly different from the approach described in Section 3 because the estimate $\hat{c}_{t'}$ for some t' in the sliding window is based on its group G which may include counts received after t' and up to time t. Thus, *Window Sum Smoother* may provide a better estimate than just using *MedianSmoother* as described in Section 3.3. We make this comparison empirically in Section 6 and we also compare against the state of the art technique for sliding window queries [2].

For event monitoring queries, the *Smoother* depends on the particular event monitor. For detecting jumps or drops, since we need the count at time t and the count received at time $t - w + 1$, we simply use the *MedianSmoother*. Actually, we may do better by also smoothing $\tilde{c}_{t-w+1}$ using its group G as defined at time t. For detecting low signal points, we need a sliding window query at each timestamp. Thus we use the *Window Sum Smoother*. Once the counts have been appropriately smoothed, we pass the estimated counts to the event monitoring functions B and f to generate an event stream (as described in Section 2.2).

5 HIERARCHICAL STREAMS

In this section, we describe an extension to PeGaSus to support queries on multiple target states as well as aggregations of states. Recall from Section 2.2.2 that the analyst can request queries on a set of states $\{s_1, \ldots, s_m\} \subseteq \mathcal{S}$ and can also ask queries about aggregations of states. We focus in particular on the setting in which the analyst has specified a hierarchy of aggregations AGG and a query on each $agg \in AGG$.

For ease of presentation, we describe our approach assuming that the analyst has requested a unit counting query on each $agg \in AGG$. However, our approach extends easily to sliding windows and event monitoring using an extension analogous to what was described in Section 4.

5.1 Hierarchical-Stream PeGaSus

First, we observe that we can answer the queries over hierarchical aggregations by simply treating each aggregation as a separate stream and running any single-stream algorithm on each input stream. Therefore, our first solution is to run PeGaSus on each stream, an algorithm we refer to as *Hierarchical-Stream PeGaSus* (HS-PGS). Formally, given a set of aggregations AGG and a corresponding set of input streams $C(agg) = c_1(agg), c_2(agg), \ldots,$ for each $agg \in AGG$, the algorithm HS-PGS executes $PeGaSus(C(agg), \frac{\epsilon}{h})$ for each $agg \in AGG$, where h is the height of AGG.

Theorem 5.1. Hierarchical-Stream PeGaSus *satisfies ϵ-differential privacy.*

Proof. The proof follows from (a) the privacy guarantee of PeGaSus (Theorem 3.1), (b) parallel composition of differential privacy across each level of the hierarchy, and (c) the sequential composition of differential privacy for each of the h hierarchy levels. □

5.2 Hierarchical-Stream PeGaSus With Pruning

We next describe an enhancement of *Hierarchical-Stream PeGaSus* that is designed to lower error when many of the input streams in the aggregation hierarchy are sparse.

The hierarchy implies a monotonicity constraint on the counts in the streams: the counts cannot decrease as one moves up the hierarchy. More formally, for any $agg_1, agg_2 \in AGG$ such that $agg_1 \subset agg_2$, then for every time step t in streams $C(agg_1)$ and $C(agg_2)$, it must be that $c_t(agg_1) \leq c_t(agg_2)$. Therefore, if the aggregated stream $C(agg_2)$ is observed to have a "small" count at time t, all the aggregated streams $C(agg_1)$ will also have small counts at time t if $agg_1 \subset agg_2$. In such cases, it will be wise to prune the count $c_t(agg_1)$ to be zero rather than consume privacy budget trying to estimate a smaller count. Pruning small counts is also beneficial because the privacy budget that would have been spent on these counts can be saved and spent on non-pruned counts.

We use this pruning idea to modify our approach as follows. At each time step, the hierarchy is traversed and streams with small counts at this time step are pruned; any streams that remain unpruned are fed into the usual PeGaSus algorithm. Note that pruning only affects the current time step; a pruned stream at time t may become unpruned at time $t + 1$.

Algorithm 4 presents our solution, which is called *Hierarchical-Stream PeGaSus with Pruning*. The function Prune (lines 1-17) describes a differentially private algorithm for pruning the hierarchy. This function is based on the idea of the *Sparse Vector Technique* [13]. Essentially, it checks each aggregation $agg \in AGG$ from level 1 to level h. If the current agg has been pruned, all its children are automatically pruned. Otherwise, we compare this aggregation's current count, $c_t(agg)$, against a user-specified threshold β (line 8). If the count is below threshold, it prunes all the children of agg. Further, the privacy budget that would have been spent on the descendants is saved (line 10). To ensure privacy, Laplace noise is added to both the count $c_t(agg)$ and the threshold β.

The *Hierarchical-Stream PeGaSus with Pruning* algorithm itself is described on lines 18-29. At each time step, it calls Prune. Then, for each aggregation, if it has been pruned, it simply outputs a count of 0 (line 21). Otherwise, it applies the PeGaSus algorithm to the stream (lines 23-25) where the privacy budget passed to *Perturber* and *Grouper* is adjusted based on what has been pruned and the height of the tree.

In our implementation, there is a small modification to *Grouper* in that we skip over past time steps that were pruned and therefore consider potentially non-contiguous groups.

To prove the privacy guarantee of Algorithm 4, we analyze the Prune function.

Theorem 5.2. *The* Prune *function in Algorithm 4 satisfies ϵ-differential privacy.*

Proof. For any two neighboring source streaming datasets, they derive neighboring multiple streams that only differ by 1 at one timestamp on one single stream. In terms of *AGG*, there is at most one list of aggregations at different levels that cover this single stream. Function Prune can be treated as an implementation of the Sparse Vector Technique from [13], which ensures ϵ-differential privacy. □

Theorem 5.3. *Algorithm 4 satisfies ϵ-differential privacy.*

Proof. For any two neighboring source streaming datasets, they derive neighboring multiple streams that only differ by 1 at one timestamp on one single stream. Calling function Prune at every timestamp with ϵ_{pr} privacy budget ensures ϵ_{pr}-differential privacy based on Theorem 5.2. For the pruned aggregated streams, we set the count to be 0, which will not leak any information. For the non-pruned aggregated streams, we apply an algorithm similar to Algorithm 1 to compute noisy counts at each timestamp. Based on the analysis of Theorem 3.1 and sequential composition of differential privacy on the aggregation set with h levels, releasing noisy counts at every timestamp will satisfy $\epsilon_p + \epsilon_g$-differential privacy. Thus, Algorithm 4 satisfies $\epsilon = \epsilon_{pr} + \epsilon_p + \epsilon_g$-differential privacy. □

Algorithm 4 *Hierarchical-Stream PeGaSus with Pruning* (PHS-PGS)

Input: *AGG* with h levels, streams $C(agg)$ for each $agg \in AGG$, privacy budget $\epsilon = \epsilon_{pr} + \epsilon_p + \epsilon_g$, threshold β

Output: Streams $\hat{C}(agg)$ for each $agg \in AGG$

```
1:  function Prune(AGG, $c_t(agg)$ for each $agg \in AGG, \epsilon, \beta$)
2:      $Pruned \leftarrow \emptyset, \epsilon_{AGG} \leftarrow \emptyset$
3:      for level $i = 1$ to $h$ do
4:          for each $agg \in AGG$ at level $i$ do
5:              if $agg \in Pruned$ then
6:                  Add every child of $agg$ to $Pruned$
7:                  Add $\epsilon_{agg} = 0$ to $\epsilon_{AGG}$
8:              else if $(c_t(agg) + Lap(2/\epsilon)) < (\beta + Lap(2/\epsilon))$ then
9:                  Add every child of $agg$ to $Pruned$
10:                 Add $\epsilon_{agg} = h - i + 1$ to $\epsilon_{AGG}$
11:             else
12:                 Add $\epsilon_{agg} = 1$ to $\epsilon_{AGG}$
13:             end if
14:         end for
15:     end for
16:     Return $Pruned, \epsilon_{AGG}$
17: end function
18: for each timestamp $t$ do
19:     $Pruned, \epsilon_{AGG} \leftarrow$ Prune($AGG, c_t(agg)$ for each $agg \in AGG, \epsilon_{pr}, \beta$)
20:     for each $agg \in AGG$ do
21:         if $agg \in Pruned$ then $\hat{c}_t(agg) \leftarrow 0$
22:         else
23:             $\tilde{c}_t(agg) \leftarrow Perturber(c_t(agg), \frac{\epsilon_{agg}(\epsilon_p+\epsilon_g)}{h} - \frac{\epsilon_g}{h})$
24:             $P_t \leftarrow Grouper(C_t(agg), P_{t-1}, \frac{\epsilon_g}{h})$
25:             $\hat{c}_t(agg) \leftarrow Smoother(\tilde{C}_t(agg), P_t)$
26:         end if
27:         Output $\hat{c}_t(agg)$
28:     end for
29: end for
```

6 EVALUATION

In this section, we evaluate the proposed algorithms on real data streams for a variety of workloads. We design the following experiments:

1. We evaluate answering unit counting queries on data streams with a single target state.
2. We evaluate answering sliding window queries on data streams with a single target state.
3. We evaluate event monitoring (detecting jumping and dropping points as well as low signal points) on data streams with a single target state.
4. We evaluate answering unit counting queries on a collection of hierarchical aggregations using data streams with multiple target states.

Dataset : Our source data comes from real traces, taken over a six month period, from approximately 4000 WiFi access points (AP) distributed across the campus of a large educational institution. We set the time interval to be 5 minutes and derive tuples from the source data. If a user u makes at least one connection to AP s within time interval t, then a tuple (u, s, t) will be added to the stream. Then, for any single AP s, a stream $C(s) = c_1(s), c_2(s), \ldots$ will be generated, where $c_t(s)$ reports the number of users who successfully connected to AP s within time interval t. For the evaluation of

Table 1: An overview of the streams derived from real WiFi access points connection traces. *Length* refers to the number of counts, each representing the number of successful connections in a 5 minute interval. *Total count* is the sum of all counts in the stream.

Stream Name	# of target states	Length	Total count
Low_5	1	57901	2846
Med_5	1	57901	101843
High_5	1	57901	2141963
Multi_5	128	20000	646254

data streams with a single target state, we pick three representative APs with different loads (called "Low_5", "Med_5" and "High_5"). For the evaluation of data streams with multiple target states, we randomly pick 128 APs (called "Multi_5"), and generate a hierarchical aggregation of these states as a binary tree of height 8. An overview of the corresponding derived streams is shown in Table 1.

6.1 Unit counting query on a single target state

Answering a unit counting query on data streams with a single target state is equivalent to releasing a private version of the entire stream. Figure 2 shows visually the real and the privately generated streams for the first 8000 timesteps of stream "High_5" under $\epsilon = 0.1$ and 0.01. We compare our PeGaSus algorithm with the *Laplace Mechanism* (LM). In PeGaSus, we set $\epsilon_p = 0.8 \times \epsilon$ and $\epsilon_g = 0.2 \times \epsilon$. We set $\theta = \frac{5}{\epsilon_g}$ in our *Grouper*. We use the *MedianSmoother* method as the *Smoother*. In each figure of a noisy stream, we truncate the negative counts to zero. Clearly, PeGaSus produces private streams that are more accurate when visualized.

We also quantitively evaluate the error of answering the unit counting query by using a couple of measures related to L_1 error. For any input data stream $C_t = \{c_1, c_2, \ldots, c_t\}$ and the output private stream $\hat{C_t} = \{\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_t\}$, the scaled total L_1 error is defined to be $\frac{\sum_{i=1}^{t} |c_i - \hat{c}_i|}{\sum_{i=1}^{t} c_i}$. We also use average L_1 error, which is defined as $\frac{\sum_{i=1}^{t} |c_i - \hat{c}_i|}{t}$.

Figure 3 reports the evaluation results on the data streams with the single target state from Table 1. In each figure, LM means *Laplace Mechanism*, and PGS is PeGaSus with the *MedianSmoother* as *Smoother*. In addition, as a simple comparison method, we use BS_t to mean a method where we do backward smoothing of results from *Laplace Mechanism*. Given a backward smoothing time k, for any $t \geq k$, $\hat{c}_k$ is updated as $\frac{\sum_{i=t-k}^{t} \hat{c}_i^{LM}}{k+1}$, where $\hat{c}_i^{LM}$ is the output from *Laplace Mechanism*. Each bar in the figures reports the scaled total L_1 error in terms of the unit counting query at all timesteps. The value is the average of 20 random trials. PeGaSus consistently performs the best on all the data streams and under both ϵ settings.

Next, we compare the impact of the three different smoothing strategies – *JSSmoother*, *MedianSmoother*, and *AverageSmoother* – in terms of answering the unit counting query. The results are shown in Figure 4. Each bar reports the $\log_{10}$ value of the average L_1 error under 20 trials in terms of each smoothing strategy. We can see that all *JSSmoother*, *MedianSmoother* and *AverageSmoother* are good smoothing strategies, but *MedianSmoother* is consistently better than *AverageSmoother* and *JSSmoother* for all data streams and ϵ settings.

6.2 Sliding window query on a single target state

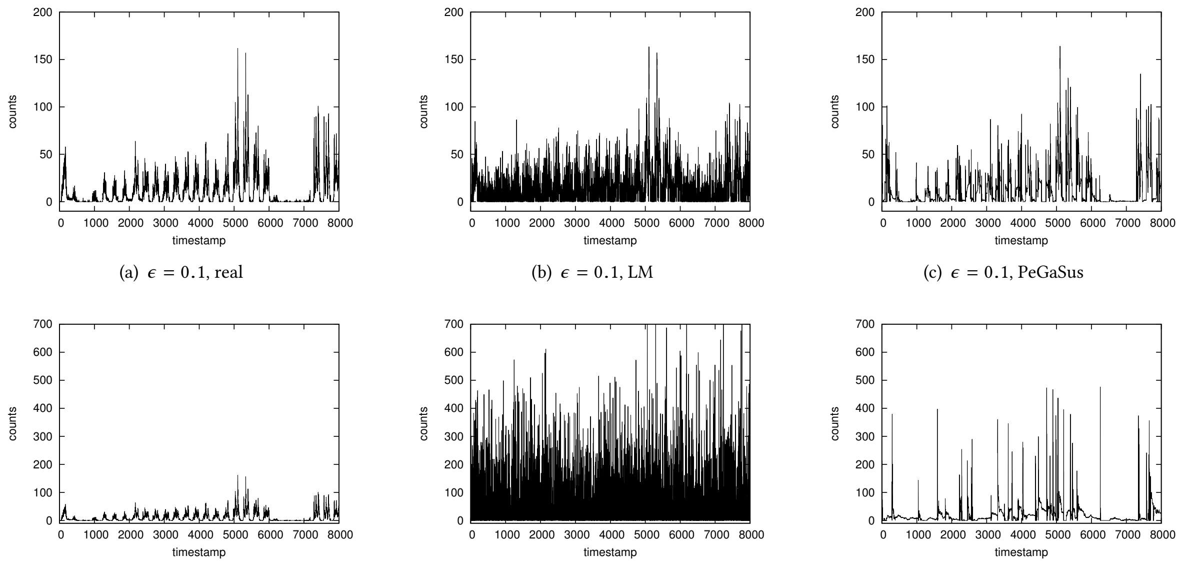
Next we evaluate the sliding window query with window size w. Figure 5 presents the results. Each point shows the average L_1 error for answering the sliding window query with size $w = 2^x$ at all timesteps and the value is the average of 20 trials. In each sub-figure, LM represents using *Laplace Mechanism* to answer the unit counting query first, then computing the sliding window query based on the noisy unit counting query answers. SW_w is the state-of-the-art data independent algorithm for computing the sliding window query in terms of a fixed window size w [4]. SW_w generates binary trees on every consecutive window of w counts and perturbs each node query of each binary tree. Then, any sliding window query with size w can be derived as the sum of the prefix and suffix of any two neighboring binary trees. PGS_MS is a variant of PeGaSus with *MedianSmoother* as *Smoother*; PGS_WWS is a variant of PeGaSus with *Window Sum Smoother* (Algorithm 3) as *Smoother*.

As shown in the figure, LM introduces excessive error. PGS_WWS performs slightly but consistently better than PGS_MS, demonstrating the benefit of using a different *Smoother* for this workload. PGS_WWS computes more accurate sliding window queries when w is not greater than 256 compared with the state of the art algorithm SW_w. When the window size becomes larger, SW_w becomes better because SW_w is designed specifically for the sliding window query with window size w while our PeGaSus may introduce a large bias into a large sliding window query. We emphasize that PeGaSus can simultaneously support all sliding window queries instead of having to split the privacy budget and design algorithms for each sliding window workload with a fixed window size.

6.3 Event monitoring on a single target state

In this experiment we consider event monitoring queries on the "High_5" data stream. Figure 6 displays the ROC curves for detecting jumping and dropping points on streams with a single target state. In each sub-figure, LM represents using *Laplace Mechanism* to generate a noisy stream and doing event monitoring on the noisy stream. PGS is our PeGaSus with *MedianSmoother*. We use a fixed window size w and threshold δ to compute the ground truth in terms of the real stream. We vary the threshold from 0 to 1000 to do the private event monitoring and compute the corresponding "True Positive Rate" and "False Positive Rate". As shown in the figures, for all different w and δ settings, when ϵ is large (= 0.1), both LM and PeGaSus perform very well and LM is slightly better than PeGaSus. However, when ϵ becomes smaller (=0.01), PeGaSus performs much better than LM.

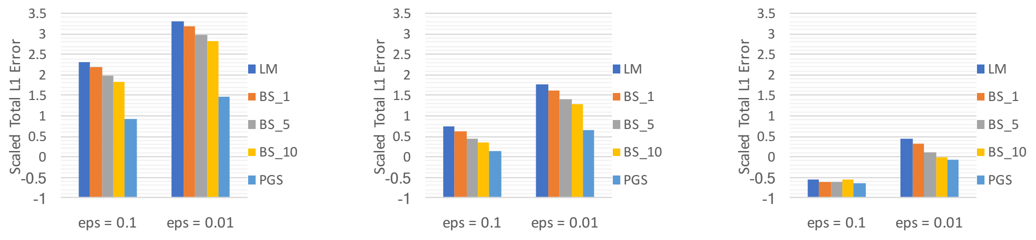
Figure 7 shows the ROC curves for detecting low signal points. Since determining the low signal points requires computing the sliding window queries, we compare SW_w from [2] with our proposed PeGaSus with Algorithm 3 as *Smoother*. We use fixed window size w and threshold δ to compute the ground truth in terms of the real stream. We vary the threshold from -4000 to 4000 to do



(a) $\epsilon = 0.1$, real (b) $\epsilon = 0.1$, LM (c) $\epsilon = 0.1$, PeGaSus

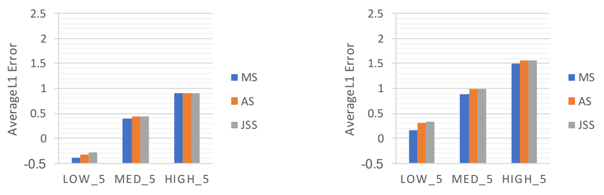
(d) $\epsilon = 0.01$, real (e) $\epsilon = 0.01$, LM (f) $\epsilon = 0.01$, PeGaSus

Figure 2: Visualizations of the "High_5" stream for 8000 timesteps. The real stream is shown on the left followed by two privately generated versions: the Laplace Mechanism (LM) and PeGaSus. Above $\epsilon = 0.1$, while below $\epsilon = 0.01$.



(a) Low_5 (b) Med_5 (c) High_5

Figure 3: Error for the unit counting query on streams with a single target state. The y-axis reports log_{10} of the scaled total L_1 error.



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.01$

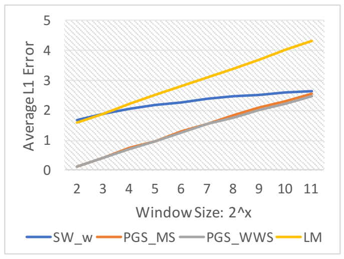
Figure 4: Smoothing strategy comparison for unit workload on various streams. "MS" is MedianSmoother, "AS" is AverageSmoother and "JSS" is JSSmoother. Y-axis reports the log_{10} of the average L_1 error.

the private event monitoring and compute the corresponding "True Positive Rate" and "False Positive Rate". As shown in the figures, for all different ϵ, w and δ settings, PeGaSus always outperforms SW_w.

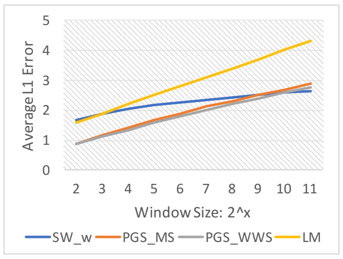
6.4 Unit counting query on hierarchical aggregated streams; multiple target states

We evaluate our proposed algorithms in Section 5 for answering unit counting query on a set of hierarchical aggregated streams among multiple target states. We use the "Multi_5" stream from Table 1 and consider a set of hierarchical aggregations AGG as a binary tree on the total number of states (128 in our case). The total levels of AGG is $\log_2(128) + 1 = 8$.

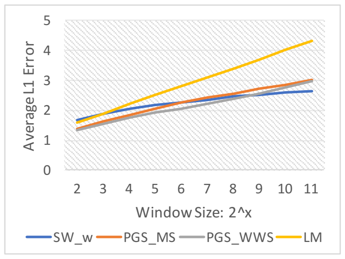
Figure 8 shows the results under two different ϵ settings. Each bar reports the $\log_{10}$ value of the average L_1 error in terms of the unit counting query on every aggregated stream at all timesteps.



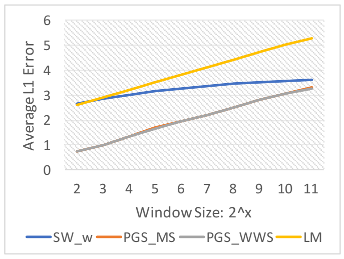
(a) Low_5, $\epsilon = 0.1$



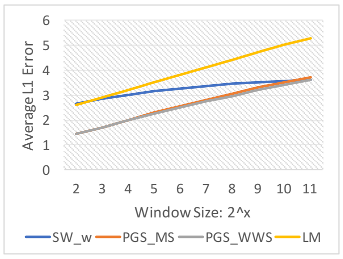
(b) Med_5, $\epsilon = 0.1$



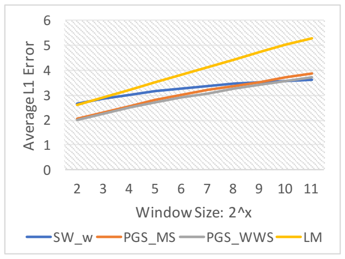
(c) High_5, $\epsilon = 0.1$



(d) Low_5, $\epsilon = 0.01$

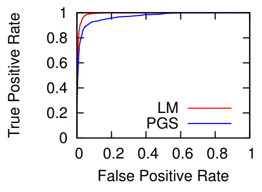


(e) Med_5, $\epsilon = 0.01$

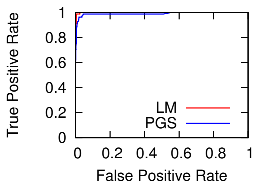


(f) High_5, $\epsilon = 0.01$

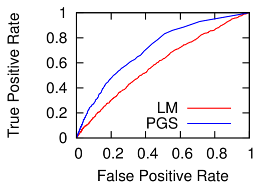
Figure 5: Error for the sliding window queries. The x-axis represents the window size as 2^x. The y-axis reports $\log_{10}$ of the average L_1 error.



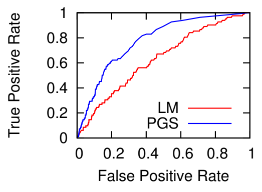
(a) $\epsilon = .1$, $w = 10$, $\delta = 100$



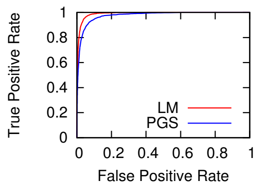
(b) $\epsilon = .1$, $w = 10$, $\delta = 150$



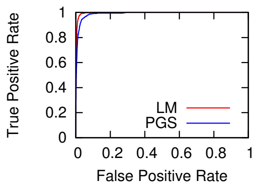
(c) $\epsilon = .01$, $w = 10$, $\delta = 100$



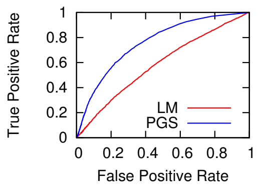
(d) $\epsilon = .01$, $w = 10$, $\delta = 150$



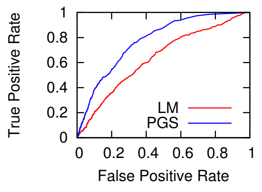
(e) $\epsilon = .1$, $w = 20$, $\delta = 100$



(f) $\epsilon = .1$, $w = 20$, $\delta = 150$



(g) $\epsilon = .01$, $w = 20$, $\delta = 100$



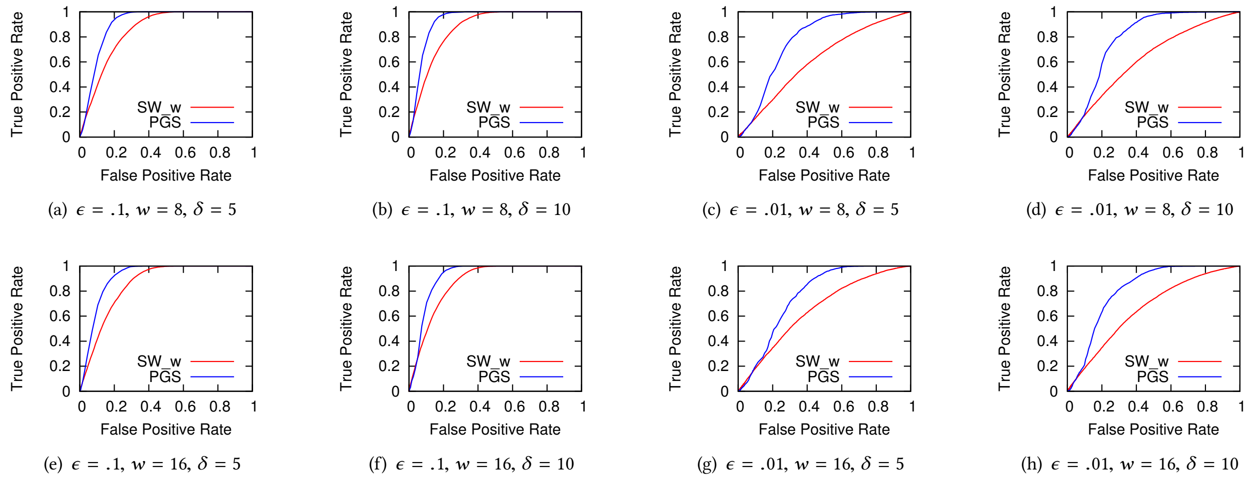
(h) $\epsilon = .01$, $w = 20$, $\delta = 150$

Figure 6: ROC curve for detecting jumping and dropping points on stream High_5. For (a), (b), (e) and (f), $\epsilon = 0.1$; For (c), (d), (g) and (h), $\epsilon = 0.01$.

The value is also the average of 20 trials. In the figure, HS-LM represents using *Laplace Mechanism* on answering the unit query on each aggregated stream with $\frac{\epsilon}{h}$ privacy budget, where h is the number of levels of *AGG*. HS-PGS is using our proposed PeGaSus with *MedianSmoother* as *Smoother* on each aggregated stream. PHS-PGS is the proposed *Hierarchical-Stream PeGaSus* with pruning (Algorithm 4). As shown in the figure, both HS-PGS and PHS-PGS reduce the average L_1 error of answering unit counting query by 1 to 2 orders of magnitude compared with the data-independent HS-LM. In addition, by doing the pruning, the utility of the results are further improved. The average L_1 error of HS-LM is over 78x (445x) times the error of PHS-PGS, and the error of HS-PGS is over 4x (9x) times the error of PHS-PGS for $\epsilon = 0.1$ (0.01).

7 RELATED WORK

There is prior work focusing on continual real-time release of aggregated statistics from streams [2, 4, 7, 15, 16]. Most of this literature focuses on releasing a single continuous query with low error under



(a) $\epsilon = .1$, $w = 8$, $\delta = 5$ (b) $\epsilon = .1$, $w = 8$, $\delta = 10$ (c) $\epsilon = .01$, $w = 8$, $\delta = 5$ (d) $\epsilon = .01$, $w = 8$, $\delta = 10$

(e) $\epsilon = .1$, $w = 16$, $\delta = 5$ (f) $\epsilon = .1$, $w = 16$, $\delta = 10$ (g) $\epsilon = .01$, $w = 16$, $\delta = 5$ (h) $\epsilon = .01$, $w = 16$, $\delta = 10$

Figure 7: ROC curve for detecting low signal points on stream High_5. For (a), (b), (e) and (f), $\epsilon = 0.1$; For (c), (d), (g) and (h), $\epsilon = 0.01$.

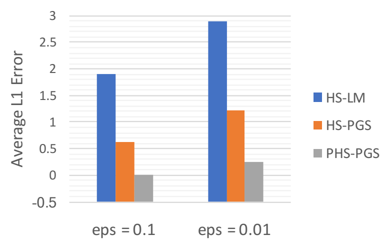


Figure 8: Evaluation of answering unit counting query on a set of hierarchical aggregated streams among multiple target states. Y-axis reports the $\log_{10}$ of the average L_1 error.

differential privacy. In contrast, PeGaSus is able to support multiple analyses on the private stream.

Fan et al. proposed *Fast* [15], an adaptive system to release real-time aggregate statistics under differential privacy by using sampling and filtering. Their algorithm is based on a different privacy model, user-level differential privacy, which is not comparable to our chosen model of event level differential privacy. In [7], Chan et al. use a privacy model that matches the one in the present work. But they focus on a single task: releasing prefix sums of the streaming data counts. For an input stream with known bounded length T, it generates a binary tree defined on the stream and perturbs the node counts of the binary tree. Then each prefix range query is computed based on a set of perturbed nodes. For streams with unbounded length, it spends half of the privacy budget on perturbing every range query between $2^i + 1$ and 2^{i+1}. Then it generates a binary tree on every sub-stream between $2^t + 1$ and 2^{t+1} for each $i = 1, 2, \ldots,$ and perturbs the node counts with the remaining half of the privacy budget. Any prefix sum at timestamp k, where $2^i <= k < 2^{i+1}$, can be computed based on the range queries between $2^i + 1$ and 2^{i+1} for all $i \leq t$ and the noisy nodes from the binary tree between $2^t + 1$ and 2^{t+1}. The authors proved that the error of each release at timestamp t is $O(\log(t))$. We did not compare with this method because we do not consider answering prefix counting queries.

Bolot et al. [2] also used a comparable privacy model, proposing an algorithm for answering sliding window queries on data streams with a fixed window size w. The basic idea is also to generate binary trees on every consecutive w data points and perturb the nodes of the binary trees. A sliding window query at each timestamp can be derived as the sum of the suffix and prefix of two consecutive binary trees. This is the state-of-the art algorithm for releasing sliding window query answers. But the algorithm is designed for any one fixed window size, which means we must split the budget for answering multiple sliding window queries with different window sizes. We compare our proposed technique with this method in Section 6 and our technique always has a better performance on our real-world WIFI dataset when answering sliding window queries with size not greater than 2^8, even if we do not split the privacy budget for different window size by using the previous method from [2]. Cao et al. [4] study a different task: answering a set of special prefix sum queries with timestamps of the form $j \times s$, where s is the step size chosen from some pre-defined step size set. The proposed algorithms sample some step sizes from the step size set and then only perturb the window queries in terms of the chosen step sizes. Then the prefix range query from the workload can be computed by composition of these perturbed window queries. We did not compare with this method since it only focuses on answering a small fixed set of prefix range queries. Kellaris et al. first proposed another privacy model called w-event differential privacy [16], which is a balance between user-level and event-level differential privacy and designed algorithms for releasing private data under w-event differential privacy.

Dwork adapted differential privacy to a continual observation setting [9], which focused on a 0/1 stream and proposed a cascading buffer counter for counting the number of 1s under event-level differential privacy. Mir et al. studied pan-private algorithms for estimating distinct count, moments and the heavy-hitter count on data streams in [21], which preserves differential privacy even if the internal memory of the algorithms is compromised. Chan et al. studied the application of monitoring the heavy hitters across a set of distributed streams [6].

Dwork et al. proposed a differentially private online partition algorithm for counting under continual observations [12]. Like our *Grouper* module, this algorithm also employs the Sparse Vector Technique [13]. However, our *Grouper* differs from Dwork et al. in the following important ways: (1) their algorithm is designed for computing partitions such that total counts of each partition are similar while ours is to find groups such that the elements in each group share similar counts. (2) Using the deviation function helps the *Grouper* to detect contiguous intervals in the stream that have a stable value (even if the values are high). On the other hand, using the total only lets us group together intervals that have counts close to zero.

8 CONCLUSION

We presented PeGaSus, a new differentially private algorithm that can simultaneously answer a variety of continuous queries at multiple resolutions on real time data streams. Our novel *Perturber*, data-adaptive *Grouper* and query specific *Smoother* approach helps release counting, sliding window and event monitoring queries with low error on sparse or stable streams. Our empirical results show that our approach outperforms state-of-the-art solutions specialized to individual queries.

There are some open questions for the future work. First, there exist some parameters in our proposed algorithms required to be set. Different setting of the parameters would affect the final outputs. Designing algorithms for data adaptive parameter tuning will be useful. Second, It will be of great usefulness to design more sophisticated *Smoother* that will go beyond recognizing only sparse or stable sub-regions. It will be even better that the *Smoother* can be adaptively adjusted in terms of the input streaming data.

Acknowledgements: This work was supported by the National Science Foundation under grants 1253327, 1408982, 1409125, 1443014, 1421325, and 1409143; and by DARPA and SPAWAR under contract N66001-15-C-4067. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A Survey. *Comput. Netw.* 54, 15 (Oct. 2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010

[2] Jean Bolot, Nadia Fawaz, S. Muthukrishnan, Aleksandar Nikolov, and Nina Taft. 2013. Private Decayed Predicate Sums on Streams. In *Proceedings of the 16th International Conference on Database Theory (ICDT '13)*. ACM, New York, NY, USA, 284–295. https://doi.org/10.1145/2448496.2448530

[3] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov. 2011. "You Might Also Like:" Privacy Risks of Collaborative Filtering. In *2011 IEEE Symposium on Security and Privacy*. 231–246. https://doi.org/10.1109/SP.2011.40

[4] Jianneng Cao, Qian Xiao, Gabriel Ghinita, Ninghui Li, Elisa Bertino, and Kian-Lee Tan. 2013. Efficient and Accurate Strategies for Differentially-private Sliding Window Queries. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 191–202. https://doi.org/10.1145/2452376.2452400

[5] Yang Cao, Masatoshi Yoshikawa, Yonghui Xiao, and Li Xiong. 2017. Quantifying Differential Privacy under Temporal Correlations. *ICDE* (2017).

[6] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. 2012. Differentially Private Continual Monitoring of Heavy Hitters from Distributed Streams. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies (PETS'12)*. Springer-Verlag, Berlin, Heidelberg, 140–159. https://doi.org/10.1007/978-3-642-31680-7_8

[7] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.* 14, 3, Article 26 (Nov. 2011), 24 pages. https://doi.org/10.1145/2043621.2043626

[8] Irit Dinur and Kobbi Nissim. 2003. Revealing Information While Preserving Privacy. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '03)*. ACM, New York, NY, USA, 202–210. https://doi.org/10.1145/773153.773173

[9] Cynthia Dwork. 2010. Differential Privacy in New Settings. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 174–183. http://dl.acm.org/citation.cfm?id=1873601.1873617

[10] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis *(TCC'06)*. Springer-Verlag, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[11] 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 (STOC '10)*. ACM, New York, NY, USA, 715–724. https://doi.org/10.1145/1806689.1806787

[12] Cynthia Dwork, Moni Naor, Omer Reingold, and Guy N. Rothblum. 2015. *Pure Differential Privacy for Rectangle Queries via Private Partitions*. Springer Berlin Heidelberg, Berlin, Heidelberg, 735–751. https://doi.org/10.1007/978-3-662-48800-3_30

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

[14] Cynthia Dwork and Sergey Yekhanin. 2008. New Efficient Attacks on Statistical Disclosure Control Mechanisms. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology (CRYPTO 2008)*. Springer-Verlag, Berlin, Heidelberg, 469–480. https://doi.org/10.1007/978-3-540-85174-5_26

[15] Liyue Fan and Li Xiong. 2012. Real-time Aggregate Monitoring with Differential Privacy. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*. ACM, New York, NY, USA, 2169–2173. https://doi.org/10.1145/2396761.2398595

[16] Georgios Kellaris, Stavros Papadopoulos, Xiaokui Xiao, and Dimitris Papadias. 2014. Differentially Private Event Sequences over Infinite Streams. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1155–1166. https://doi.org/10.14778/2732977.2732989

[17] 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 (SIGMOD '11)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1989323.1989345

[18] Daniel Kifer and Ashwin Machanavajjhala. 2014. Pufferfish: A Framework for Mathematical Privacy Definitions. *ACM Trans. Database Syst.* 39, 1, Article 3 (Jan. 2014), 36 pages. https://doi.org/10.1145/2514689

[19] Chao Li, Michael Hay, Gerome Miklau, and Yue Wang. 2014. A Data- and Workload-Aware Query Answering Algorithm for Range Queries Under Differential Privacy. *PVLDB* 7, 5 (2014), 341–352. http://www.vldb.org/pvldb/vol7/p341-li.pdf

[20] Changchang Liu, Supriyo Chakraborty, and Prateek Mittal. 2016. Dependence Makes You Vulnerable: Differential Privacy Under Dependent Tuples. In *NDSS*.

[21] Darakhshan Mir, S. Muthukrishnan, Aleksandar Nikolov, and Rebecca N. Wright. 2011. Pan-private Algorithms via Statistics on Sketches. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '11)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1989284.1989290

[22] Shuang Song, Yizhen Wang, and Kamalika Chaudhuri. 2017. Pufferfish Privacy Mechanisms for Correlated Data. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1291–1306. https://doi.org/10.1145/3035918.3064025

[23] Charles Stein. 1956. Inadmissibility of the Usual Estimator for the Mean of a Multivariate Normal Distribution. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. University of California Press, 197–206. http://projecteuclid.org/euclid.bsmsp/1200501656

[24] Yonghui Xiao and Li Xiong. 2015. Protecting Locations with Differential Privacy Under Temporal Correlations. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1298–1309. https://doi.org/10.1145/2810103.2813640