# Efficient and Accurate Strategies for Differentially-Private Sliding Window Queries

Jianneng Cao
Purdue University
caojn@purdue.edu

Qian Xiao
National University of Singapore
xiaoqian@nus.edu.sg

Gabriel Ghinita
University of Massachusetts Boston
gabriel.ghinita@umb.edu

Ninghui Li
Purdue University
ninghui@cs.purdue.edu

Elisa Bertino
Purdue University
bertino@purdue.edu

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

## ABSTRACT

Regularly releasing the aggregate statistics about data streams in a privacy-preserving way not only serves valuable commercial and social purposes, but also protects the privacy of individuals. This problem has already been studied under differential privacy, but only for the case of a single continuous query that covers the entire time span, e.g., counting the number of tuples seen so far in the stream. However, most real-world applications are *window-based*, that is, they are interested in the statistical information about streaming data within a window, instead of the whole unbound stream. Furthermore, a Data Stream Management System (DSMS) may need to answer numerous correlated aggregated queries simultaneously, rather than a single one. To cope with these requirements, we study how to release differentially private answers for a set of *sliding window* aggregate queries. We propose two solutions, each consisting of query *sampling* and *composition*. We first selectively sample a subset of representative sliding window queries from the set of all the submitted ones. The representative queries are answered by adding Laplace noises in a way satisfying differential privacy. For each non-representative query, we compose its answer from the query results of those representatives. The experimental evaluation shows that our solutions are efficient and effective.

## 1. INTRODUCTION

Nowadays, data streams are common to many applications, such as online transactions, disease monitoring, environment sensing, and financial fraudulence detection. Typically, they arrive at high speed continuously, and usually are unbounded. Online processing of such data brings unique commercial opportunities to the companies. Regularly releasing aggregate statistics about them also serves valuable social purposes. However, data streams may contain sensitive personal information, e.g., a stream about patients with a highly infectious disease. Aggregate results released from such a stream can help health department better control the disease, but on the other hand, the query results about the sensitive information may potentially put the individual privacy at risk.

Differential privacy [12] has become the de facto standard notion of privacy. In this paper, we adopt it to protect the privacy of individuals, whose data appear in the stream. Informally, differential privacy requires that the output of aggregated queries be approximately the same, even if any single tuple in the input database is arbitrarily modified. Such a concept ensures that the adversarial attacks against a person are essentially equally likely to occur, whether that person's tuple is in the database or not. Till now, most approaches [6, 19, 26, 20, 32, 22, 17, 31] proposed to guarantee differential privacy are developed for static datasets. As such, they cannot be directly applied to data streams, which are usually unbounded, transient, and require query results to be updated when new data arrive.

Recently, differential privacy has also been investigated in the context of data streams. As far as we know, all the proposed methods [11, 13, 8, 14] consider a single aggregate query (i.e., a counter), which differential-privately releases the number of 1's seen so far in a stream. These methods are insufficient for real-world applications for the following two major reasons. Firstly, most real-world applications are *window-based*. Consider a binary stream about whether individuals are affected by a highly contagious disease. It is more likely that health organizations are interested in the statistics about the streaming data within a most recent window (e.g., the number of affected persons in *last 10 days*), instead of the whole unbounded stream. Secondly, a Data Stream Management System (DSMS) may need to answer numerous correlated aggregate queries simultaneously, instead of just one. In the above running example, the organizations may submit queries with various *window size* and *step size* (step size specifies window update frequency, e.g., update the number of affected persons every *2 days*).

To cope with the unique requirements posed by data streams, in this work we propose a differentially-private framework customized for sliding window count queries. It operates in two phases. First, it samples a subset of representative queries from all the submitted ones. The representatives are answered by adding Laplace noises in a way satisfying differential privacy. For each non-representative query, we compose its answer from the query results of those representatives. Our framework answers directly only a subset of selected representative queries. Such a strategy allows the magnitude of added Laplace noises to be lowered. Thus, the returned query results can be more accurate. We develop two solutions from our framework. The former studies a special case,


*EDBT/ICDT* '13, March 18 - 22 2013, Genoa, Italy

in which the step size (i.e., window update frequency) of a bigger query is always a multiple of that of a smaller one. The latter investigates the general case, in which a step size can be an arbitrary positive integer. Our extensive experimental results show that both solutions achieve query accuracy superior to existing approaches.

The remainder of the paper is organized as follows. The next section presents fundamental definitions and background knowledge. We propose the solution for the special case in Section 3, and that for the general case in Section 4. Section 5 reports the experimental results. We survey related work in Section 6, and conclude the paper in Section 7.

# 2. FUNDAMENTAL DEFINITIONS AND BACKGROUND

## 2.1 Query Model

Similar to existing solutions [13, 8] on differential privacy for data streams, we consider an unbounded stream of append-only tuples $\mathcal{DS}[t]$, where time $t \in \{1, 2, 3, \dots\}$ and each tuple has a binary value 0 or 1. However, instead of considering a single query that counts the number of 1's seen so far as in [13, 8], we investigate a more flexible model, where a set of continuous *sliding-window* count queries are registered. This is more practical, since real-world applications are interested in most recent information, rather than the whole unbounded stream. Each query is defined by three parameters: *registration time*, *window size*, and *step size*. For simplicity, we assume that the registration time is the same for all the queries. Table 1 summarizes the notations used.

**Table 1: Summary of Notations**

| Symbol | Denotation |
|---|---|
| $\mathcal{DS}$ | Data stream with binary-valued tuples (0's and 1's) |
| $\Gamma$ | Set of submitted sliding window count queries |
| $\Gamma_R$ | Set of *representative* sliding window count queries |
| $\mathcal{L}$ | Set of all the distinct steps in $\Gamma$ |
| $\mathcal{L}_R$ | Set of all the distinct steps in $\Gamma_R$ |
| $Q[W, S]$ | Query with window $W$ and step $S$ |
| $S_i^j$ | The $j^{th}$ slot of step $S_i$ |
| $\mathcal{CW}$ | Set of cycle windows |

Let $Q[W, S]$ be a continuous query with window $W$ and step size $S$. We assume that window size is a multiple of step size, i.e., $W = j \times S$, where $j$ is a positive integer. Figure 1 shows 4 queries $Q_1[W_1, S_1]$, $Q_2[W_2, S_1]$, $Q_3[W_3, S_2]$, and $Q_4[W_4, S_3]$, where $Q_1$ and $Q_2$ share the same step $S_1$. The size relationship between a window and its step is also illustrated, e.g., $W_2$ is three times as big as its step $S_1$.

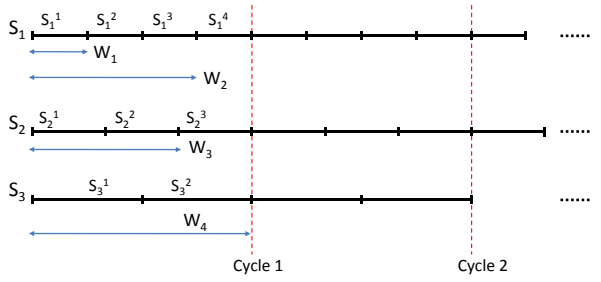



**Figure 1: Steps, windows and cycles.**

DEFINITION 1 (SLOT). *Let $\mathcal{DS}$ be a data stream, and $S$ be the step of a query. We partition $\mathcal{DS}$ into segments, each having size $S$. These segments are the* slots *of step $S$.*

In Figure 1, $S_1^1, S_1^2, S_1^3, S_1^4, \dots$ are the slots of step $S_1$. Similarly, $S_2^1, S_2^2, S_2^3, \dots$ are the slots of step $S_2$, and $S_3^1, S_3^2, \dots$ are those of step $S_3$. Essentially, a slot is a time interval; a step's slots form a partition of $\mathcal{DS}$. Given slot $S_i^j$, we denote its *begin* time by $S_i^j.b$, and its *end* time by $S_i^j.e$.

For each query $Q[W, S]$, its window $W$ slides forward at every time interval of $S$. Thus, a sequence of windows is generated. To answer $Q$, the Data Stream Management System (DSMS) returns the number of 1's within each window. Take query $Q_3[W_3, S_2]$ as an example: when $W_3$ slides forward, windows $W_3^1, W_3^2, W_3^3, W_3^4, \dots$ are generated (Figure 2) and the number of 1's in each window is counted. Just like a slot, a window $W$ represents a time interval. Similarly, we use $W.b$ and $W.e$ to define the *begin* and *end* time of $W$.

In a DSMS with multiple continuous queries, each query may slide forward at a different time interval. However, after a certain time period, the system will return to its *initial* state. In Figure 1, after window $W_1$ and $W_2$ slide 4 times, $W_3$ slides 3 times, and $W_4$ slides 2 times, the DSMS configuration will be the same as at the origin of time, with the only difference that it is shifted forward by one *cycle*. The concept of cycle is formally defined as follows:

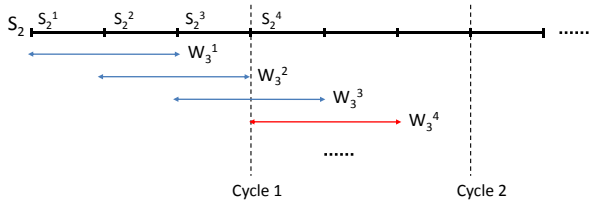



**Figure 2: Sliding window and its cycles**

DEFINITION 2 (CYCLE). *Denote by $\Gamma$ the set of continuous queries in a DSMS. Let $t_1$ be a timestamp when the begin times of all sliding window queries in $\Gamma$ coincide, and let $t_2 > t_1$ be the minimum timestamp when all the windows have again the same begin time. Then, the time interval $[t_1, t_2]$ is a cycle for $\Gamma$, and its length is $t_2 - t_1$.*

The cycle length is determined by the step sizes in $\Gamma$. Given $\ell$ distinct steps $S_1, S_2, \dots, S_\ell$ in $\Gamma$, the cycle length is the *Least Common Multiple* $\mathsf{LCM}(S_1, S_2, \dots, S_\ell)$. Let the lengths of the three steps $S_1$, $S_2$, and $S_3$ in Figure 1 be 3, 4, and 6, respectively. Then, the cycle length is $\mathsf{LCM}(3, 4, 6) = 12$.

The repetitive nature of cycles allows us to characterize the DSMS behavior by studying only the first cycle. Thus, we will focus on sliding windows only in the first cycle (others are just shifted forward from them by one or more cycles).

DEFINITION 3 (CYCLE WINDOW SET). *The cycle window set $\mathcal{CW}$ is the set of all the sliding windows of queries in $\Gamma$, for which the window begin time falls within the first cycle of $\Gamma$.*

Figure 1 shows 4 queries; all the sliding windows with their begin times falling in Cycle 1 form the cycle window set. In particular, for query $Q_3$, the set includes $W_3^1$, $W_3^2$, and $W_3^3$ (Figure 2). Window $W_3^4$ (not in the set) is similar to $W_3^1$, but shifted one cycle forward.

## 2.2 Privacy Model

We adopt the state-of-the-art differential privacy model [12], summarized next.

DEFINITION 4 (NEIGHBORING STREAMS[8]). *Data streams $\mathcal{DS}_1$ and $\mathcal{DS}_2$ are* neighboring, *if they differ only at one timestamp $x$, that is, $\mathcal{DS}_1[x] \neq \mathcal{DS}_2[x]$ and $\mathcal{DS}_1[t] = \mathcal{DS}_2[t]$ for $t \in \{1, 2, \dots\} \setminus \{x\}$.*

Differential privacy ensures that an adversary cannot distinguish two neighboring streams based on the query results on them.

DEFINITION 5 (DIFFERENTIAL PRIVACY [8]). *A random mechanism $\mathcal{R}$ satisfies $\epsilon$-differential privacy, if for any pair of neighboring streams $(\mathcal{DS}_1, \mathcal{DS}_2)$ and any measurable subset $O$ in the query output domain, $Pr[\mathcal{R}(\mathcal{DS}_1) \in O] \leq e^{\epsilon} \cdot Pr[\mathcal{R}(\mathcal{DS}_2) \in O]$.*

The $L_1$ *sensitivity* [12] of a set of functions $F$ is defined as

$$\mathsf{Sen}(F) = \max_{\forall \mathcal{DS}_1, \mathcal{DS}_2} \left( \sum_{f \in F} |f(\mathcal{DS}_1) - f(\mathcal{DS}_2)| \right) \quad (1)$$

where $\mathcal{DS}_1$ and $\mathcal{DS}_2$ are neighboring streams.

Dwork et al [12] prove that differential privacy can be achieved by adding to each query answer a noise generated according to Laplace distribution. Specifically, the noise is a variable $\mathsf{Lap}(\lambda)$ with probability density function $Pr[\mathsf{Lap}(\lambda) = x] = \frac{1}{2\lambda} e^{-|x|/\lambda}$. $\mathsf{Lap}(\lambda)$ has mean 0 and variance $2\lambda^2$. Furthermore, a random mechanism complies with $\epsilon$-differential privacy, if it adds to each function $f \in F$ independent Laplace noise $\mathsf{Lap}(\lambda)$, where $\lambda = \frac{\mathsf{Sen}(F)}{\epsilon}$.

In our model, window size is a multiple of its step size. Thus, any sliding window can be derived by a concatenation of the slots of its step. In Figure 2, window $W_3^1$ is the concatenation of $S_2^1$ and $S_2^2$, whereas window $W_3^2$ is the concatenation of $S_2^2$ and $S_2^3$. Therefore, instead of answering each window query directly (i.e., count the number of 1's in the window), we can first answer each *slot query* (i.e., count the number of 1's in the slot), and then compose the window query answer by the slot query answers.

Suppose that $\mathcal{L}$ is the set of steps in $\Gamma$. Then, all the slots of all the steps in $\mathcal{L}$ form $F$. Although the cardinality of $F$ is infinite for an unbounded stream $\mathcal{DS}$, changing the value of a streaming tuple at any timestamp in $\mathcal{DS}$ will change the answers of exactly $|\mathcal{L}|$ slot queries. Thus, if we refer to the definition of sensitivity in Eq. (1), $\mathsf{Sen}(F) = |\mathcal{L}|$. If we add to each slot query Laplace noise $\mathsf{Lap}(\frac{|\mathcal{L}|}{\epsilon})$, then $\epsilon$-differential privacy is satisfied. Consequently, the noise for a window will be the summation of the noises of its composite slots.

In Figure 1, $\mathcal{L} = \{S_1, S_2, S_3\}$ and we add independent Laplace noise $\mathsf{Lap}(\frac{3}{\epsilon})$ to each slot query. Consider query $Q_3$. Its first window $W_3^1$ in Cycle 1 is a concatenation of slots $S_2^1$ and $S_2^2$. Hence, the query answer for $W_3^1$ is the summation of answers to $S_2^1$ and $S_2^2$. Accordingly, the added noise for the window is $\mathsf{Lap}(\frac{3}{\epsilon}) + \mathsf{Lap}(\frac{3}{\epsilon})$.

## 2.3 Query Accuracy

Adding noise is necessary for privacy, but it reduces query accuracy. Let $act$ and $approx$ be the actual and approximate (i.e., noisy) query answers for a sliding window $W$, respectively. As in prior work [17, 9], we adopt variance $\mathsf{Var}(W) = \mathbb{E}[(act - approx)^2]$ to measure the accuracy of the query answer, where $\mathbb{E}$ denotes mathematical expectation. Let $Q[W, S]$ be a query, and let $\mathcal{CW}^Q = \{W^1, W^2, W^3, \dots\}$ be the set of windows of $Q$ that fall within the cycle window set $\mathcal{CW}$. We define the error of $Q$ with respect to $\mathcal{CW}$ as

$$\mathsf{Err}(Q, \mathcal{CW}) = \frac{\sum_{\forall W^i \in \mathcal{CW}^Q} \mathsf{Var}(W^i)}{|\mathcal{CW}^Q|},$$

Due to the repetitive nature of cycles, the error of $Q$ is independent of any specific cycle, so we simplify the notation $\mathsf{Err}(Q, \mathcal{CW})$ to $\mathsf{Err}(Q)$. Given a set of queries $\Gamma$, their workload error is $\mathsf{WorkloadErr}(\Gamma) = \sum_{Q \in \Gamma} \mathsf{Err}(Q)$. Our objective is to ensure privacy while minimizing workload error.

# 3. A SPECIAL CASE STUDY

In this section, we present a solution for a special case, where all the steps satisfy a special relationship, namely the size of a bigger step is a *multiple* of that of a smaller one. Formally, denote by $S_1, S_2, \dots, S_{|\mathcal{L}|}$ all the steps in $\mathcal{L}$ sorted in ascending order of their sizes. The special relationship among steps is represented as $S_{i+1} = \ell_i \times S_i$, where $i = 1, 2, \dots, |\mathcal{L}| - 1$, and $\ell_i$ is a positive integer. The special case ensures that any sliding window with a bigger step size can always be exactly expressed as a concatenation of the slots of a smaller step.

Let $\Gamma$ be the set of all the submitted sliding window count queries. If we choose to answer each of them in a standard way, then the sensitivity is $|\mathcal{L}|$, where $\mathcal{L}$ is the set of all the steps in $\Gamma$, and the added Laplace noise for each slot query is $\mathsf{Lap}(\frac{|\mathcal{L}|}{\epsilon})$. Instead, we can sample $\Gamma$ to select a set of representative queries $\Gamma_R \subset \Gamma$, and answer them only. Assume $\mathcal{L}_R$ is the set of all the steps in $\Gamma_R$. Now, the added Laplace noise for any slot query of a step in $\mathcal{L}_R$ is $\mathsf{Lap}(\frac{|\mathcal{L}_R|}{\epsilon})$. Clearly, the noise is reduced. Furthermore, we can also see that the representative query set $\Gamma_R$ and its corresponding set of steps $\mathcal{L}_R$ is a *one-to-one correspondence*. Once $\mathcal{L}_R$ is calculated, $\Gamma_R$ is automatically determined. Therefore, in the following we will investigate the sampling of steps.

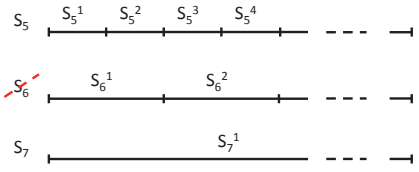



**Figure 3: A motivating example**

EXAMPLE 1. *Consider 3 queries $Q_5[W_5, S_5]$, $Q_6[W_6, S_6]$, and $Q_7[W_7, S_7]$, with window (step) sizes of 15, 20, and 350 (5, 10, and 350), respectively. As such, we have a cycle for them as shown in Figure 3. If we choose to answer each of them, then the sensitivity is 3, and the variance for each slot query is $2 \cdot \mathbb{E}[(\mathsf{Lap}(\frac{3}{\epsilon}))^2] = \frac{18}{\epsilon^2}$. The first sliding window $W_5^1$ of $Q_5$ is composed of three slots $S_5^1$, $S_5^2$, and $S_5^3$, each having a variance of $18/\epsilon^2$. Hence, $\mathsf{Var}(W_5^1) = 3 \times (18/\epsilon^2) = 54/\epsilon^2$. Since the variance of each sliding window of $Q_5$ is the same, $\mathsf{Err}(Q_5) = 54/\epsilon^2$. Similarly, we obtain $\mathsf{Err}(Q_6) = 36/\epsilon^2$ and $\mathsf{Err}(Q_7) = 18/\epsilon^2$. Alternatively, suppose that we choose to answer only queries $Q_5$ and $Q_7$. Then, the sensitivity is reduced to $|\mathcal{L}_R| = |\{S_5, S_7\}| = 2$, and the variance of any slot query for $S_5$ and $S_7$ is $2 \cdot \mathbb{E}[(\mathsf{Lap}(\frac{2}{\epsilon}))^2] = \frac{8}{\epsilon^2}$. Accordingly, $\mathsf{Err}(Q_5) = 24/\epsilon^2$ and $\mathsf{Err}(Q_7) = 8/\epsilon^2$. Still, the query answer for $Q_6$ can be derived by slot queries of step $S_5$. In particular, the first sliding window of $Q_6$ originally composed of slots $S_6^1$ and $S_6^2$ is now derivable from $S_5^1$, $S_5^2$, $S_5^3$, and $S_5^4$. Thus, $\mathsf{Err}(Q_6) = 4 \cdot \frac{8}{\epsilon^2} = \frac{32}{\epsilon^2}$. Obviously, the query errors are smaller.*

Motivated by the above example, we present a strategy to answer count queries in a differentially private way. It consists of two phases: **sampling** and **composition**.

*Sampling*. We partition the step list $S_1, S_2, \ldots, S_{|\mathcal{L}|}$ into a set of sub-lists $L_S = \{sl_1, sl_2, \ldots, sl_k\}$, such that $sl_1 = \{S_1, S_2, \ldots, S_{j_1}\}$, $sl_2 = \{S_{j_1+1}, S_{j_1+2}, \ldots, S_{j_2}\}$, $\ldots$, and $sl_k = \{S_{j_{k-1}+1}, S_{j_{k-1}+2}, \ldots, S_{j_k}\}$ ($j_k = |\mathcal{L}|$). From each sub-list in $L_S$, we sample its first step (i.e., the smallest one) and insert it into $\mathcal{L}_R$. Thus, $\mathcal{L}_R = \{S_1, S_{j_1+1}, \ldots, S_{j_{k-1}+1}\}$. The slot queries of all the steps in $\mathcal{L}_R$ are answered by adding Laplace noise $\mathsf{Lap}(|\mathcal{L}_R|/\epsilon)$, where $|\mathcal{L}_R| = k$. However, for any of the remaining steps (i.e., $\mathcal{L} \setminus \mathcal{L}_R$), none of its slot queries is answered. Consider Example 1. The three steps $S_5$, $S_6$, $S_7$ are partitioned into two sub-lists $\{S_5, S_6\}$ and $\{S_7\}$. Steps $S_5$ in the first sub-list and $S_7$ in the second are chosen to be representatives, and their slot queries are answered by adding Laplace noise $\mathsf{Lap}(2/\epsilon)$.

*Composition*. Given any continuous query $Q[W, S]$, no matter whether its step $S$ is sampled into $\mathcal{L}_R$ or not, we locate the sub-list containing $S$. Without loss of generality, assume that $sl = \{S_i, S_{i+1}, \ldots, S_j\}$ ($i \leq j$) is the sub-list in $L_S$ containing $S$. Then, $S_i \in \mathcal{L}_R$, and all the slot queries of $S_i$ are answered. Since $S$ is a multiple of $S_i$, any window $W$ of query $Q$ can be composed by the slots of $S_i$. Continue Example 1. Step $S_6$ of query $Q_6$ is not selected to be a representative. However, $S_6$ falls in the first sub-list $\{S_5, S_6\}$. Hence, we can use slots of $S_5$ to compose each window of $Q_6$.

The first phase *sampling* partitions $\mathcal{L}$ into a set of sub-lists $L_S = \{sl_1, sl_2, \ldots, sl_k\}$. Given sub-list $sl_i$ ($i = 1, 2, \ldots, k$), let $lq_i$ be all the submitted queries with their steps falling in $sl_i$. Then, $L_Q = \{lq_1, lq_2, \ldots, lq_k\}$ is a partition of the set of all the submitted queries $\Gamma$. We can compute the workload error of $\Gamma$ as follows. Let $sl = \{S_i, S_{i+1}, \ldots, S_j\}$ be a sub-list in $L_S$. Then, $S_i \in \mathcal{L}_R$. Suppose that $lq \in L_Q$ is the set of queries with their steps falling in $sl$. For each query $Q[W, S] \in lq$, its sliding window is composed of exactly $W/S_i$ slots of the representative step $S_i$. So the error of $Q$ is $\mathsf{Err}(Q) = 2(k/\epsilon)^2 \times (W/S_i)$, and the error of all the queries in $lq$ is

$$\mathsf{Err}^{\mathsf{sl}}(i, j, k) = \sum_{\forall Q[W,S] \in lq \ \wedge \ S \in sl} \mathsf{Err}(Q). \tag{2}$$

Accordingly, the summation of the errors regarding all $lq \in L_Q$ (or correspondingly $sl \in L_S$) is the workload error of $\Gamma$.

We adopt dynamic programming to sample the steps, so that the workload error is minimized. Let $S_1, S_2, \ldots, S_{|\mathcal{L}|}$ be the list of all the steps in $\mathcal{L}$ sorted in the ascending order of their sizes. Suppose that $k = |\mathcal{L}_R|$ steps are to be sampled from them. Let $S_1$, $S_2$, $\ldots$, $S_j$ be the prefix of $j$ steps of $\mathcal{L}$. Suppose that the prefix is to be partitioned into $n$ sub-lists. Then, the minimal error regarding these $n$ sub-lists is

$$g(j, n, k) = \min_{n \leq i \leq j} \{g(i-1, n-1, k) + \mathsf{Err}^{\mathsf{sl}}(i, j, k)\}. \tag{3}$$

On the right of Equation 3, steps $S_i, S_{i+1}, \ldots, S_j$ are assigned to a sub-list, and step $S_i$ is picked as a representative, while a recursion is applied on steps $S_1$, $S_2$, $\ldots$, $S_{i-1}$ to partition them into $n-1$ sub-lists. The cutting position $i$ is so decided, that the error is minimized. If $n = 1$, then $g(j, n, k) = \mathsf{Err}^{\mathsf{sl}}(1, j, k)$, that is, the whole prefix forms a single sub-list.

Function SampleDP adopts *memoization* [21] (i.e., a dynamic programming approach) to elaborate Equation 3. Lines 1-3 handle the case, in which the prefix (i.e., $S_1, S_2, \ldots, S_j$) forms a single sub-list. In this case, the first step is selected (line 2). Point, a 3-D array, stores the indices of the sampled steps. Lines 6-8 splits the prefix into two parts: $\{S_1, S_2, \ldots, S_{i-1}\}$, and $\{S_i, S_{i+1}, \ldots, S_j\}$. The latter part composes a sub-list (line 8), while a recursion is applied on the former to split it into $n-1$ sub-lists (lines 6-7). Line 6 tries to reuse previous results to split the former part (see Function Reuse below). $V$, another 3-D array with each element initialized to $\infty$, records the errors, that is, $V[j][n][k] = g(j, n, k)$. Lines 5-12 iteratively test each position $i$, and record the one that minimizes the error.

**Function** SampleDP ($j$, $n$, $k$)

```
1  if n == 1 then
2      Point[j][n][k] = 1;
3      Return Err^sl(1, j, k);
4  else
5      for i = n to j do
6          if Reuse(i − 1, n − 1, k) == false then
7              V[i−1][n−1][k] = SampleDP(i−1, n−1, k);
8          tmp = V[i − 1][n − 1][k] + Err^sl(i, j, k);
9          if tmp < V[j][n][k] then
10             V[j][n][k] = tmp;
11             Point[j][n][k] = i;
12     Return V[j][n][k];
```

For a given value of $k$, we call $\mathsf{SampleDP}(|\mathcal{L}|, k, k)$ to partition $\mathcal{L}$ into $k$ sub-lists, and obtain a minimal workload error with respect to this specific $k$ value. The value of $k$ varies from 1 to $|\mathcal{L}|$; the one minimizing workload error is chosen

$$\mathsf{WorkloadErr}(\Gamma) = \min_{k=1}^{|\mathcal{L}|} \{\mathsf{SampleDP}(|\mathcal{L}|, k, k)\}. \tag{4}$$

Suppose that $k = \alpha$ produces the minimum workload error. Then, $\mathcal{L}_R = \{S_{c_1}, S_{c_2}, \ldots, S_{c_\alpha}\}$, where $c_\alpha = \mathsf{Point}[|\mathcal{L}|][\alpha][\alpha]$, $c_{\alpha-1} = \mathsf{Point}[c_\alpha - 1][\alpha - 1][\alpha]$, $\ldots$, $c_2 = \mathsf{Point}[c_3 - 1][2][\alpha]$, and $c_1 = \mathsf{Point}[c_2 - 1][1][\alpha] = 1$.

When calculating $\mathsf{SampleDP}(|\mathcal{L}|, k, k)$, the number of all the possible recursions is at most $\sum_{j=1}^{|\mathcal{L}|} \sum_{n=1}^{k} 1 = |\mathcal{L}| \cdot k$. In each recursion, $j$-$n$+1 tests are enumerated (lines 5-11). Hence, the time complexity of $\mathsf{SampleDP}(|\mathcal{L}|, k, k)$ is $\sum_{j=1}^{|\mathcal{L}|} \sum_{n=1}^{k} (j - n + 1)$. Value $k$ varies from 1 to $|\mathcal{L}|$ to minimize workload error. So the time complexity of Equation 4 is $\sum_{k=1}^{|\mathcal{L}|} \sum_{j=1}^{|\mathcal{L}|} \sum_{n=1}^{k} (j - n + 1)$, upper bounded by $O(|\mathcal{L}|^4)$.

According to Equation 4, Function SampleDP ($j$, $n$, $k$) needs to be called $|\mathcal{L}|$ times, each for a different $k$. However, the theorem below tells that for any two values $n \leq k_1, k_2 \leq |\mathcal{L}|$, we can derive the output of SampleDP ($j$, $n$, $k_2$) by *reusing* that of SampleDP ($j$, $n$, $k_1$). That is, we need to call only one of these two functions. Therefore, the efficiency of finding the minimal workload error can be improved.

THEOREM 1. *Let $S_1, S_2, \ldots, S_j$ be the prefix of $j$ steps of $\mathcal{L}$, and $n$ be the number of resultant sub-lists by a partitioning on the prefix. Given any two positive integers $n \leq k_1, k_2 \leq |\mathcal{L}|$, then $\frac{g(j,n,k_1)}{g(j,n,k_2)} = \left(\frac{k_1}{k_2}\right)^2$.*

PROOF. We prove the theorem by contradiction. Assume that $\frac{g(j,n,k_1)}{g(j,n,k_2)} \neq \left(\frac{k_1}{k_2}\right)^2$. Let $sl_1$, $sl_2$, $\ldots$, $sl_n$ be the resultant sub-lists created by $g(j, n, k_1)$, and $\min(sl_i)$ be the smallest step in $sl_i$ selected into $\mathcal{L}_R$, where $i = 1, 2, \ldots, n$. Suppose that $lq_i = \{Q[W, S] | S \in sl_i\}$ is the set of queries with their steps falling in

$sl_i$. By Equation 2, the error of $lq_i$ is

$$2(k_1/\epsilon)^2 \cdot \sum_{\forall Q[W,S] \in lq_i} (W/\min(sl_i)).$$

Hence, the error for $\bigcup_{i=1}^{n} lq_i$ is

$$g(j,n,k_1) = 2(k_1/\epsilon)^2 \cdot \sum_{i=1}^{n} \left( \sum_{\forall Q[W,S] \in lq_i} (W/\min(sl_i)) \right).$$

Denote $\sum_{i=1}^{n} \sum_{\forall Q[W,S] \in lq_i} (W/\min(sl_i))$ by $X$, then

$$g(j,n,k_1) = 2(k_1/\epsilon)^2 \cdot X. \quad (5)$$

Similarly, let $\overline{sl_1}$, $\overline{sl_2}$, ..., $\overline{sl_n}$ be the resultant sub-lists created by $g(j,n,k_2)$, step $\min(\overline{sl_i})$ be the smallest one in sub-list $\overline{sl_i}$, and $\overline{lq_i} = \{Q[W,S] | S \in \overline{sl_i}\}$. Then, the error for $\bigcup_{i=1}^{n} \overline{lq_i}$ is

$$g(j,n,k_2) = 2(k_2/\epsilon)^2 \cdot \sum_{i=1}^{n} \left( \sum_{\forall Q[W,S] \in \overline{lq_i}} (W/\min(\overline{sl_i})) \right).$$

Query set $\bigcup_{i=1}^{n} \overline{lq_i}$ is equal to $\bigcup_{i=1}^{n} lq_i$, since both of them are the set of queries with their steps in $\{S_1, S_2, \ldots, S_j\}$. Denote $\sum_{i=1}^{n} \sum_{\forall Q[W,S] \in \overline{lq_i}} (W/\min(\overline{sl_i}))$ by $Y$, then

$$g(j,n,k_2) = 2(k_2/\epsilon)^2 \cdot Y. \quad (6)$$

Since $\frac{g(j,n,k_1)}{g(j,n,k_2)} \neq \left(\frac{k_1}{k_2}\right)^2$, we have $X \neq Y$ (by Equations 5 and 6). Without loss of generality, suppose $X > Y$. As a consequence, if $g(j,n,k_1)$ partitions $S_1, S_2, \ldots, S_j$ into $\overline{sl_1}$, $\overline{sl_2}$, ..., $\overline{sl_n}$, then the error for $\bigcup_{i=1}^{n} lq_i$ is reduced to

$$(g(j,n,k_1) = 2(k_1/\epsilon)^2 \cdot Y) < 2(k_1/\epsilon)^2 \cdot X, \quad (7)$$

which contradicts Equation 3. □

**Function** Reuse ($i$, $j$, $k$)

```
1 if V[i][j][k] ≠ ∞ then
2     Return true;
3 for ℓ = j to k − 1 do
4     if V[i][j][ℓ] ≠ ∞ then
5         V[i][j][k] = V[i][j][ℓ] × (k/ℓ)^2;
6         Return true;
7 Return false;
```

Theorem 1 essentially says that the $n$ resultant sub-lists are independent of any specific value of $k$. Based on it, we implement Function Reuse. If $g(i,j,k)$ does not exist, it checks if there is a previous call of $g(i,j,\ell)$, where $j \leq \ell < k$ (lines 3-4). If this is the case, then $g(i,j,k)$ is computed efficiently by reusing the output of $g(i,j,\ell)$ (line 5).

# 4. THE GENERAL CASE FRAMEWORK

In this section, we will investigate the general case, in which a step size can be an arbitrary positive integer. We preserve the two-phase strategy structure used in the special case, namely sample a set of representative steps, and then use them to answer all the sliding window queries. For the general case, the set of representative steps that we choose is 'similar' to the set of all the distinct steps. The similarity is measured by Earth Mover's Distance (EMD) [28]. We describe the sampling phase in Section 4.1 and the composition phase in Section 4.2. In Section 4.3 we provide a unified view of the entire query answering strategy.

## 4.1 The Step Sampling

Suppose that there are $m_i$ queries with step $S_i$ in $\Gamma$, $i$ = 1, 2, ..., $|\mathcal{L}|$. We define $m_i$ as the weight of the step, and define the *weighted step distribution* in $\Gamma$ by

$$\mathcal{P} = (p_1, p_2, \ldots, p_{|\mathcal{L}|}) = \left( \frac{m_1}{\sum_{i=1}^{|\mathcal{L}|} m_i}, \frac{m_2}{\sum_{i=1}^{|\mathcal{L}|} m_i}, \ldots, \frac{m_{|\mathcal{L}|}}{\sum_{i=1}^{|\mathcal{L}|} m_i} \right),$$

where $p_i = \frac{m_i}{\sum_{i=1}^{|\mathcal{L}|} m_i}$ is the normalized distribution of step $S_i$ in $\Gamma$. If we decide that step $S_i$ be sampled into $\mathcal{L}_R$, then the $m_i$ queries with this step size are put into $\Gamma_R$. Assume that the sampled steps are $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$. Then, we obtain another weighted step distribution

$$\begin{aligned} \mathcal{Q} &= (\ldots, q_{i_1}, \ldots, q_{i_2}, \ldots, q_{i_k}, \ldots) \\ &= \left( \ldots, \frac{m_{i_1}}{\sum_{\ell=1}^{k} m_{i_\ell}}, \ldots, \frac{m_{i_2}}{\sum_{\ell=1}^{k} m_{i_\ell}}, \ldots, \frac{m_{i_k}}{\sum_{\ell=1}^{k} m_{i_\ell}}, \ldots \right), \end{aligned}$$

where $q_{i_\ell}$ ($\ell = 1, 2, \ldots, k$) is the normalized distribution of step $S_{i_\ell}$ in $\Gamma_R$ and $q_i = 0$ for any $i \notin \{i_1, i_2, \ldots, i_k\}$ . In Figure 1 we have 2 queries with step $S_1$, 1 query with step $S_2$, and 1 with step $S_3$. Thus, we have the weighted step distribution $\mathcal{P} = \left(\frac{2}{4}, \frac{1}{4}, \frac{1}{4}\right)$. In the sampling, if we select steps $S_1$ and $S_3$, then $\mathcal{Q} = \left(\frac{2}{3}, 0, \frac{1}{3}\right)$.

We need a distance function to measure the difference between $\mathcal{P}$ and $\mathcal{Q}$ to guide our sampling procedure. Here, we adopt Earth Mover's Distance (EMD) [28], which considers the semantic relationship among steps (see below). Please refer to [24] for a detailed discussion about the advantages of EMD over other distance metrics (e.g., KL-divergence).

### 4.1.1 Earth Mover's Distance

The *Earth Mover's Distance* (EMD) is suggested as a metric for quantifying the difference between distributions. Intuitively, it views one distribution as a mass of *earth* piles spread over a space, and the other as a collection of *holes*, in which the mass fits, over the same space. The EMD between the two is defined as the minimum *work* needed to fill the holes with earth, thereby *transforming* one distribution to the other.

Let $\mathcal{P} = (p_1, p_2, \ldots, p_{|\mathcal{L}|})$ be the distribution of "holes", $\mathcal{Q} = (q_1, q_2, \ldots, q_{|\mathcal{L}|})$ that of "earth", $d_{ij}$ the *ground distance* of $q_i$ from $p_j$, and $F = [f_{ij}], f_{ij} \geq 0$ a flow of mass of earth moved from element $q_i$ to $p_j$, $1 \leq i, j \leq |\mathcal{L}|$. The EMD is the *minimum* value of the work required to transform $\mathcal{Q}$ to $\mathcal{P}$ by $F$:

$$WORK(\mathcal{P}, \mathcal{Q}, F) = \sum_{i=1}^{|\mathcal{L}|} \sum_{j=1}^{|\mathcal{L}|} d_{ij} \times f_{ij}$$

Let $S_1, S_2, \ldots, S_{|\mathcal{L}|}$ be the set of the steps of all the queries. Without loss of generality, we assume that the steps are sorted in the ascending order of their lengths, i.e., $S_j > S_i$ when $j > i$. We define the ground distance between step pair $S_i$ and $S_j$, $1 \leq i < j \leq |\mathcal{L}|$, as follows:

$$d_{ij} = \frac{S_j - S_i}{S_{|\mathcal{L}|} - S_1} \quad (8)$$

Based on the defined ground distance above, the minimal work for transforming $\mathcal{Q}$ to $\mathcal{P}$ can be calculated by sequentially satisfying the *earth* needs of each *hole* element, moving earth from/to its immediate neighbor pile. Thus, the EMD between $\mathcal{P}$ and $\mathcal{Q}$ is defined as:

$$EMD(\mathcal{P}, \mathcal{Q}) = \sum_{i=1}^{|\mathcal{L}|-1} \frac{S_{i+1} - S_i}{S_{|\mathcal{L}|} - S_1} \cdot \left| \sum_{j=1}^{i} (q_j - p_j) \right| \quad (9)$$

EXAMPLE 2. *Suppose that the three steps $S_1$, $S_2$, and $S_3$ in Figure 1 have the lengths* 3, 4, *and* 6, *respectively. Then, the ground distance $d_{12}$ between $S_1$ and $S_2$ is $\frac{4-3}{6-3} = \frac{1}{3}$, $d_{23} = \frac{2}{3}$, and $d_{13} = 1$. $\mathcal{P} = (p_1, p_2, p_3) = \left(\frac{2}{4}, \frac{1}{4}, \frac{1}{4}\right)$. Assume that $\mathcal{L}_R = \{S_1, S_3\}$. Then, $\Gamma_R = \{Q_1, Q_2, Q_4\}$, and $\mathcal{Q} = (q_1, q_2, q_3) = \left(\frac{2}{3}, 0, \frac{1}{3}\right)$. The EMD between $\mathcal{P}$ and $\mathcal{Q}$ is calculated as follows. Let $\mathcal{P}$ be the set of holes, and $\mathcal{Q}$ be the piles of earth. We fill hole $p_1$ by earth $q_1$ (cost is* 0, *since $d_{11} = 0$). The extra earth $q_1 - p_1 = 2/3 - 2/4 = 1/6$ is moved to pile $q_2$ with a cost of $d_{12} \cdot |q_1 - p_1| = \frac{1}{18}$. Earth pile $q_2$ is now with earth $1/6$, and it needs to acquire an amount of earth $|(q_1 + q_2) - (p_1 + p_2)| = 3/4 - 2/3 = 1/12$ from $q_3$ with a cost of $d_{23} \cdot |(q_1 + q_2) - (p_1 + p_2)| = \frac{1}{18}$. After that, $q_2$ fills $p_2$ (cost is* 0), *and $q_3$ fills $p_3$ (cost is 0). Therefore, $EMD(\mathcal{P}, \mathcal{Q}) = d_{12} \cdot |q_1 - p_1| + d_{23} \cdot |(q_1 + q_2) - (p_1 + p_2)| = \frac{1}{9}$. In a similar way, we can find $EMD(\mathcal{P}, \mathcal{Q}') = \frac{2}{9}$.*

### 4.1.2 The Algorithm

**Function** SampleEMD ($\Gamma$, $\delta$)

```
1  Store all the steps of Γ into L;
2  Sort the steps of L in ascending order of length;
3  Let P be the weighted step distribution in Γ;
4  Let C be a sorted list, initialize to {0, |L|};
5  Γ_R = Select(Γ, L, C);
6  Let Q be the weighted step distribution in Γ_R;
7  while EMD(P, Q) > δ do
8      Update(Γ_R, L, P, C);
9      Re-compute Q according to Γ_R;
10 Return Γ_R;
```

Function SampleEMD takes as input all the submitted queries $\Gamma$ and a distance threshold $\delta$, and outputs a set of representative queries, whose weighted step distribution differs from that in $\Gamma$ by at most $\delta$. The set of steps in $\Gamma$ is stored in $\mathcal{L} = \{S_1, S_2, \ldots, S_{|\mathcal{L}|}\}$ in ascending order of length (lines 1-2). Function Select chooses representative queries from $\Gamma$ into $\Gamma_R$. In particular, list $C$ stores a set of sorted cutting positions. For any two consecutive positions $i_1$ and $i_2$ ($i_1 < i_2$) of $C$, all steps $S_j$ with $i_1 < j \leq i_2$ are pushed into one group. A step is picked from each group. Among all the steps in the group, the one with the maximum weight (i.e., the number of queries with this step size) is chosen. If there is a tie, we pick the one with the maximum length. Once a step is picked, all the queries in $\Gamma$ with this step are pushed into $\Gamma_R$. Lines 4-5 initialize $\Gamma_R$ by taking all the steps in $\mathcal{L}$ as a single group. Then, SampleEMD iteratively cuts $\mathcal{L}$ into smaller groups, and updates $\Gamma_R$ correspondingly (lines 7-9). In each round, Function Update chooses a new cutting position, which splits an existing group into two (line 8). Based on the new set of groups, $\Gamma_R$ is updated, and $\mathcal{Q}$ is re-computed (line 9). The iteration terminates when the EMD between the distribution in $\Gamma$ and that in $\Gamma_R$ is at most $\delta$ (line 7).

Procedure Update tests on all the possible cutting positions (lines 2-9), and finds the one (line 10), which minimizes the EMD between $\mathcal{P}$ and $\mathcal{Q}$. Lines 3-4 ignore all the positions that have already been chosen for the group partitioning in $\mathcal{L}$. Consider each valid cutting position $i$ ($1 \leq i \leq |\mathcal{L}|$). If it is added to $C$ (i.e., $C \cup \{i\}$), then $\mathcal{L}$ would be partitioned into a new set of groups, and Function Select would determine another set of representative queries (line 5). The EMD between $\mathcal{P}$ and the weighted step distribution for the representatives is calculated (line 7). For each possible position $i$, the one that allows minimum EMD is recorded (lines 7-9). Finally, line 10 inserts this position into $C$, and $\Gamma_R$ is recalculated correspondingly (line 11).

**Procedure** Update ($\Gamma_R$, $\mathcal{L}$, $\mathcal{P}$, $C$)

```
1  d^min = ∞;
2  for i ← 1 to |L| − 1 do
3      if i ∈ C then
4          Continue;
5      Γ^i = Select(Γ, L, C ∪ {i});
6      Let Q^i be the weighted step distribution in Γ^i;
7      if d^min > EMD(P, Q^i) then
8          d^min = EMD(P, Q^i);
9          pos = i;
10 Insert pos into C;
11 Γ_R = Select(Γ, L, C);
```

The time complexity of Function SampleEMD is dominated by the loop (lines 7-9). Since there are all together $|\mathcal{L}| - 1$ cutting positions, the loop runs at most $|\mathcal{L}| - 1$ times. If all the cutting positions are chosen into $C$, then the representative queries will be exactly the same as the submitted queries and the EMD will be 0. This also indicates that SampleEMD will finally terminates. In each round of the loop, Procedure Update is called and a new cutting position is found. Hence, in the $i$-th round, Update only needs to test $|\mathcal{L}| - i$ positions that have not been selected. When testing one of these available positions, Functions Select and EMD are called (both of them have a time complexity linear in $|\mathcal{L}|$). Therefore, the time complexity of SampleEMD is upper bounded by $\sum_{i=1}^{|\mathcal{L}|-1} \sum_{\ell=1}^{|\mathcal{L}|-i} |\mathcal{L}|$, that is, $O(|\mathcal{L}|^3)$.

EXAMPLE 3. *Once again we examine the query processing system in Figure 1. As in Example 2, we assume that the three steps $S_1$, $S_2$, and $S_3$ are of lengths* 3, 4, *and* 6, *respectively. Then, $\mathcal{P} = \left(\frac{2}{4}, \frac{1}{4}, \frac{1}{4}\right)$, and $\mathcal{L} = \{S_1, S_2, S_3\}$ (lines 1-3, Function* SampleEMD*). Let $\delta = 0.2$. Without any cutting of $\mathcal{L}$, we will select $S_1$ ($S_1$'s weight is biggest; lines 4-5). Thus, $\mathcal{Q} = (1, 0, 0)$, and $EMD(\mathcal{P}, \mathcal{Q}) = 1/3$. Since $1/3 > \delta$, we need to partition $\mathcal{L}$ by calling Procedure* Update*. Possible cutting position $i = 1$ partitions $\mathcal{L}$ into two groups $\mathcal{G}_1 = \{S_1\}$ and $\mathcal{G}_2 = \{S_2, S_3\}$. The selected step from $\mathcal{G}_1$ is $S_1$, and that from $\mathcal{G}_2$ is $S_3$ ($S_3$ has a longer length). Hence, $\mathcal{Q}^1 = \left(\frac{2}{3}, 0, \frac{1}{3}\right)$. If we choose the cutting position to be at $i = 2$, then coincidentally, $\mathcal{Q}^2 = \mathcal{Q}^1$. The EMD between $\mathcal{P}$ and $\mathcal{Q}^1$ (or $\mathcal{Q}^2$) is $1/9$. Therefore, $pos = 1$ is chosen to be the cutting position (line 10, Procedure* Update*). After the cutting, the EMD between $\mathcal{P}$ and $\mathcal{Q}$ is smaller than $\delta$. Therefore, queries with steps $S_1$ and $S_3$ are selected as representatives. Furthermore, if $\delta < 1/9$, then Procedure* Update *should be called again, and $pos = 2$ will be chosen, indicating that $\mathcal{G}_2$ needs to be further split into $\{S_2\}$ and $\{S_3\}$. Consequently, all the queries would be selected as representatives.*

## 4.2 The Query Composition

In Section 4.1 we have determined the set of representative queries $\Gamma_R$, and thus also $\mathcal{L}_R$ the set of steps in $\Gamma_R$. In the following, given any window $W$, no matter whether it belongs to a representative query in $\Gamma_R$ or not, we will adopt a dynamic programming procedure to compute a set of slots, which belong to representative steps, to compose $W$. We start from the calculation of cumulative noise for $W$ from its composite slots[1]:

$$f(W.b, W.e) = \min\{\mathsf{Noise}(S) + f(S.e + 1, W.e) | \\ S \in \mathcal{L}_R \wedge S.b = W.b\} \qquad (10)$$

[1]We will use a step to represent any of its slots.

Any slot $S \in \mathcal{L}_R$, whose *begin* time is equal to that of window $W$ (i.e., $S.b = W.b$), is a possible candidate to partially cover (or compose) $W$. After that, the remaining part of $W$, not yet covered, is a time interval $[S.e + 1, W.e]$, whose covering (or composition) becomes an independent sub-problem and thus can be done recursively. If $S$ is adopted, a certain noise quantified by $\mathsf{Noise}(S) = \mathsf{Lap}(\frac{|\mathcal{L}_R|}{\epsilon})$ needs to be accumulated. In sum, the total noise is $\mathsf{Noise}(S) + f(S.e, W.e)$. We iterate all possible solutions, and the one with the minimal cumulative noise is chosen.

**Function** Compose ($B$, $E$)

1 **if** $B == E$ **then**
2  $V_f[B] = 0$;
3  Return $V_f[B]$;
4 **foreach** *slot* $S \in \mathcal{L}_R$ *with* $S.b == B$ **do**
5  **if** $V_f[S.e + 1] == \infty$ **then**
6  $V_f[S.e + 1] = \mathsf{Compose}(S.e + 1, E)$;
7  $tmp = \mathsf{Noise}(S) + V_f[S.e + 1]$;
8  **if** $tmp < V_f[B]$ **then**
9  $V_f[B] = tmp$;
10  $Cover[B] = S$;
11 Return $V_f[B]$;

Function Compose employs *memoization* [21] to find the minimal cumulative noise for a window $W$. We use an array $V_f$ to record noises. In particular, $V_f[W.b]$ is to record the minimal cumulative noise for a window (or a time span), starting at $W.b$ and ending at $W.e$, similarly, element $V_f[W.b + 1]$ will store the minimum noise for a time span $[W.b + 1, W.e]$, and so on. Before the algorithm is called, we initialize each element in the array to be $\infty$. The algorithm's input parameters $B$ and $E$ represent the start and end times of a window, respectively, and they will be set to $W.b$ and $W.e$ when it is called. Given a window with size equal to 0, no noise will be added (lines 1-3). Line 4 selects a slot $S$ that starts at $B$. Partially covering window $W$ by $S$ incurs noise $\mathsf{Noise}(S)$. Function Compose is recursively called to calculate the minimum noise for $W$'s non-covered part, which starts at $S.e + 1$ and ends at $E$ (lines 5-7). In addition, we memorize the solution for this sub-problem by line 6 for a later *reuse*. If slot $S$ is selected to compose $W$, then the total noise will be a summation of $\mathsf{Noise}(S)$ and the noise of the sub-problem (line 7). If the summation demands a smaller noise than the current solution (line 8), then it is recorded (line 9), and the corresponding slot $S$ is buffered (line 10). After all the possible slots with *begin* time equal to $B$ are tested, we obtain the optimal solution (lines 4-10). Finally, line 11 returns the minimal noise.

Function Compose also buffers in array $Cover$ the group of slots, which composes the window $W$. In particular, they are $st_1 = Cover[W.b]$, $st_2 = Cover[st_1.e + 1]$, $st_3 = Cover[st_2.e + 1]$, ..., and $st_k = Cover[st_{k-1}.e + 1]$, where $st_k.e = W.e$.

The time complexity of Function Compose is $O(n \cdot W)$, where $W$ is the window size and $n$ is the number of slots falling in $W$. Its analysis is similar to that of *knapsack problem* [21].

## 4.3 The Complete Strategy

We summarize our framework by combining Section 4.1 and Section 4.2. Because of the step sampling, the slot queries of the non-representative steps are not answered. Still, for any sliding window $W$ of a continuous query $Q$, no matter whether its step is selected as a representative or not, we need to find a group of slots to compose $W$. To make sure such a composition is always possible, we split some slots by *injection*.

DEFINITION 6 (SLOT INJECTION). *Let $S_1$ and $S_2$ be two distinct steps. Let $S_1^i$ and $S_2^j$ be any pair of slots, belonging to $S_1$ and $S_2$, respectively, such that $S_2^j.e \in (S_1^i.b, S_1^i.e)$. We say that the slots of $S_2$ are injected into those of $S_1$, if we split $S_1^i$ into $S_1^{i,1}$ and $S_1^{i,2}$, so that $[S_1^{i,1}.b, S_1^{i,1}.e] = [S_1^i.b, S_2^j.e]$ and $[S_1^{i,2}.b, S_1^{i,2}.e] = [S_2^j.e + 1, S_1^i.e]$.*

Consider the query processing system in Figure 1. Let $\mathcal{L}_R = \{S_1, S_3\}$ be the set of representative steps. Suppose that the slots of non-representative step $S_2$ are injected into those of $S_1$ (Figure 4), such that $S_1^{2,1}.e = S_2^1.e$, $S_1^{2,2}.b = S_2^2.b$, $S_1^{3,1}.e = S_2^2.e$, and $S_1^{3,2}.b = S_2^3.b$. Now window $W_3^1$, originally composed of slots $S_2^1$ and $S_2^2$, can be derived by slots $S_3^1$ and $S_1^{3,1}$. Note that after the slot injection, some slots of a step are split, thus they do not have the same size as the step. Still, we will say that they are the slots of the step.

**Algorithm:** QueryPlan($\Gamma$)

1 Initialize $Covers$ be to an empty set;
2 Set $min^{wl} = \infty$;
3 **for** $\delta = 0$; $\delta < 1$; $\delta = \delta + incr$ **do**
4  $\Gamma_R = \mathsf{SampleEMD}(\Gamma, \delta)$;
5  Let $\mathcal{L}$ ($\mathcal{L}_R$) be all the steps in $\Gamma$ ($\Gamma_R$);
6  Find the shortest step $SS$ in $\mathcal{L}_R$;
7  $\mathsf{Inject}(SS, \mathcal{L} \setminus \mathcal{L}_R)$;
8  Calculate $\mathbb{C}_1$, the first cycle of $\Gamma$;
9  $\mathcal{CW} = \mathsf{CycleWindow}(\mathbb{C}_1)$;
10  Set $tmp^c$ to be empty;
11  **foreach** *window* $W \in \mathcal{CW}$ **do**
12  $\mathsf{Compose}(W.b, W.e)$;
13  Buffer in $tmp^c$ the slots composing $W$;
14  Let $tmp^{wl}$ be the workload error of $\Gamma$ in this round;
15  **if** $min^{wl} > tmp^{wl}$ **then**
16  $min^{wl} = tmp^{wl}$;
17  $Covers = tmp^c$;

Algorithm QueryPlan combines Functions SampleEMD and Compose together to calculate the composite slots for each sliding window in a cycle. Given the set of submitted queries $\Gamma$ and an EMD threshold $\delta$, it first calls Function SampleEMD to determine the set of representative steps and the queries containing them (line 4 ). Line 6 finds the shortest representative step $SS$. Line 7 performs slot injection, so that all the slots, which belong to non-representative steps (i.e., $\mathcal{L} \setminus \mathcal{L}_R$), are injected into those of $SS$. Line 7 is to ensure that we can always find a group of slots to exactly compose each window that belongs to a non-representative query. Lines 8-9 compute the cycle windows for the first cycle $\mathbb{C}_1$ in the stream. We find the composite slots for each window in $\mathbb{C}_1$ (lines 11-13), and calculate the workload error of the queries (line 14).

The value of $\delta$ decides the set of representative steps, and accordingly determines the workload error for the queries. Therefore, we enumerate $\delta$ from 0 to 1, each time increasing it by $incr$ (lines 3-17). In the experiments, we set $incr = 0.1$. The minimum workload error is stored in $min^{wl}$ (lines 2, 14-16). The composite slots for the sliding windows are stored in $Covers$ (lines 1, 17).

EXAMPLE 4. *Figure 1 is a query processing system with queries $\Gamma = \{Q_1, Q_2, Q_3, Q_4\}$ and steps $\mathcal{L} = \{S_1, S_2, S_3\}$. Consider the iteration when $\delta = 0.2$. According to Example 3, the sampling procedure returns $\mathcal{L}_R = \{S_1, S_3\}$. Hence, $\Gamma_R = \{Q_1, Q_2, Q_4\}$. We inject the slots of $S_2$ (not appearing in $\mathcal{L}_R$) into*

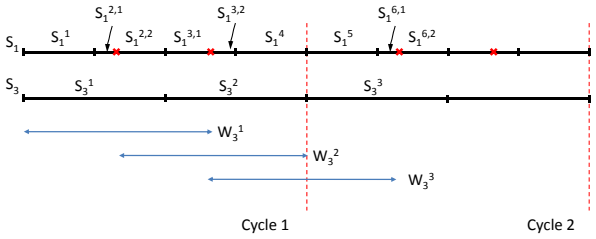



**Figure 4: An example of window composition**

*those of $S_1$ ($S_1$ has the minimum length in $\mathcal{L}_R$) as illustrated in Figure 4. The cycle windows of Cycle 1 contain all the windows whose start times fall in the interval of Cycle 1. We need to calculate the composite slots for each window in Cycle 1. In particular, we show the calculation for those of query $Q_3$ (the processing of sliding windows for other queries is similar and omitted). The optimal solution with the minimum noise says that window $W_3^1$ is composed by $\{S_3^1, S_1^{3,1}\}$ with the error $2\times(2/\epsilon)^2$, window $W_3^2$ by $\{S_1^{2,2}, S_3^2\}$ with the error $2\times(2/\epsilon)^2$, and $W_3^3$ by $\{S_1^{3,2}, S_1^4, S_1^5, S_1^{6,1}\}$ with the error $4\times(2/\epsilon)^2$.*

# 5. PERFORMANCE EVALUATION

In this section we evaluate our schemes. Our prototypes were implemented by Python, and the experiments ran on an Intel Core Quad 2.83 GHz CPU with 4GB RAM running Windows 7. We denote our solution for the special case (Section 3) by *DP*, since it adopts dynamic programming to sample representative steps. We use *EMD* to represent our solution for the general case (Section 4), because it chooses Earth Mover's Distance [28] to obtain the set of representative steps. In addition, we also implemented two benchmarks, *Base* and *Binary*. *Base* is a naive scheme, which answers the slot queries of all the steps.

*Binary* is an adaptation of the approach in [8] to handle sliding window queries[2]. It splits the data stream into segments of $T$ tuples each, where $T$ is a threshold. A binary tree structure is then built over the tuples in each segment. A leaf node in the tree represents a sequence of binary values. In the experiments, we set the sequence length (i.e., the leaf size) to 10, which is equal to the minimum step size. The number of 1's in the sequence is recorded in the leaf node. A non-leaf node counts the number of 1's in all the leaves under the subtree rooted at itself. Obviously, the sensitivity of such a structure is proportional to the tree depth $\lceil\log(T/10)\rceil$, and Laplace noise $\mathsf{Lap}(\frac{\lceil\log(T/10)\rceil+1}{\epsilon})$ is added to the count of each node. Given any sliding window, we compose its answer by selecting a minimum number of nodes from the binary trees. By default, we set $T$ to the size of the largest sliding window among all the queries.

We created three data streams. (a) *uniform*– this is a synthetic stream with 5,000,000 values (0's or 1's) generated from a uniform (Bernoulli) distribution. (b) *adult*– it contains 32,561 values derived from the raw dataset [1] as follows: a tuple in the raw dataset is mapped to 0 if its salary value $\leq 50K$, and 1 otherwise. (c) *census*– it has 500,000 numbers created from another raw dataset [2] by mapping raw tuples with the salary value $\geq 25k$ to 1's, and all the others to 0's. Unless otherwise stated, the default stream used is *census*. To measure the accuracy of the query answers, we employ three utility metrics. The first is the *workload error* as defined in Section 2. The other two metrics are *absolute error* = $|approx - act|$ and *relative error* = $|approx - act|/act$, where $approx$ and $act$ are the released approximate count value and the actual count value for a query, respectively. Furthermore, since Laplace noises are random, we repeat each experiment 10 times, and report the average results.

[2]Similar approach is proposed in [7] to study private sums on decayed streams.

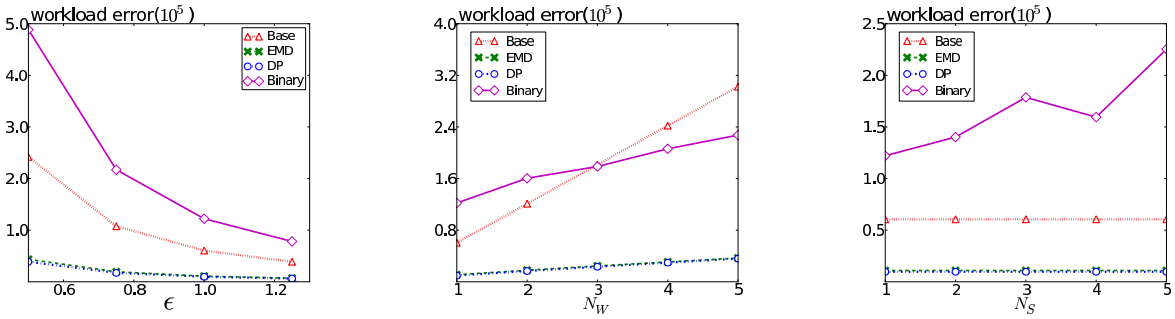
## 5.1 Special Queries

In this section, we study queries for the special case, where the size of a bigger step is a multiple of that of a smaller one. *EMD*, our general method, can process queries with any step sizes. Thus, the experiments also include it. We first create a sequence of 10 steps $20, 40, 80, \ldots, 10240$, such that for any two consecutive steps in the sequence the bigger one is twice the size of the smaller one. Then, for each query $Q[W, S]$, its step size $S$ is selected randomly from the set $\{20, 40, 80, \ldots, 10240\}$ (i.e., steps in the sequence), and its window size $W$ is set to $j \times S$, where $j$ is randomly chosen from $[1, 10]$. In this way, we generate 100 queries.

We first investigate the effect of $\epsilon$ on the workload errors of the four involved schemes. From Figure 5 (a), we observe that all schemes behave in a similar way – the workload error deceases as $\epsilon$ increases. This means as $\epsilon$ increases, a lower magnitude of Laplace noise would be added into answers; it also means that lower degree of privacy would be guaranteed. This validates the fact that the lower the degree of privacy guaranteed, the lower the error incurred. From the results, it is also clear that our proposed approaches, *DP* and *EMD*, perform almost equally effective while consistently incurring lower errors than *Base* and *Binary* at the same level of privacy guarantee. This is expected as both *Base* and *Binary* have much bigger sensitivity.

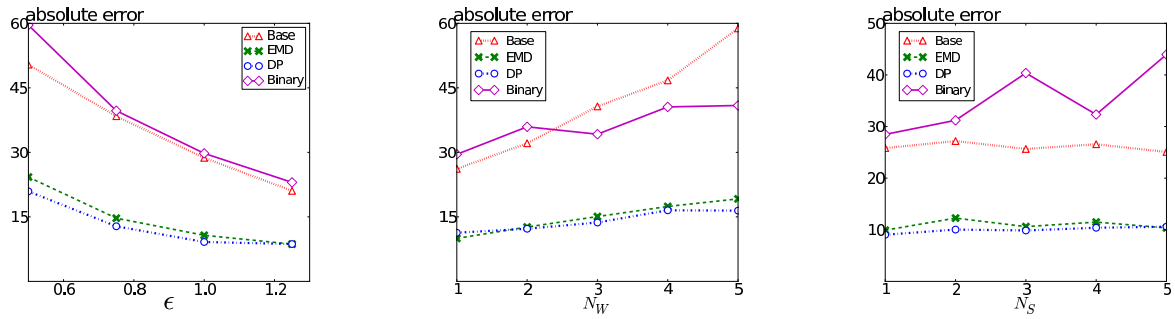
Figure 5 (b) compares the four methods as we fix $\epsilon = 1.0$ and vary window size. The window size of a query $Q[W, S]$ is equal to $j \times S$, where $j$ is an integer falling in $[1, 10]$. We vary the window size by increasing it to $N_W \times j \times S$, where $N_W$ ranges from 1 to 5. The results show that the workload error for all schemes increases as a function of $N_W$. This is because as window size becomes larger, more slots are needed to compose the window and hence more noises are added.

Next, we examine the effect of step size on workload error. Again, we set $\epsilon$ to be 1.0. For each query $Q[W, S]$, we increase its step size to $N_S \times S$, where $N_S$ is from 1 to 5. Figure 5 (c) reports the results. As step sizes increase by varying $N_S$, the Laplace noises added to the slot queries of the steps stay the same. Thus, the error of each sliding window, which is composed of slots, does not change either. Consequently, the workload errors of the three schemes, i.e., *DP*, *EMD*, and *Base*, are independent of $N_S$. However, as step size increases, the segment size $T$ (equal to the size of the biggest sliding window) of the *Binary* approach also increases. Hence, the sensitivity $\lceil\log(T/10)\rceil + 1$ for *Binary* becomes larger. Therefore, in general its workload error increases as a function of $N_S$.




(a) varying $\epsilon$ (b) varying window size (c) varying step size

**Figure 5: Workload error (special case)**




(a) varying $\epsilon$ (b) varying window size (c) varying step size

**Figure 6: Average absolute error (special case)**

Our next experiment examines absolute error. Figure 6 is the results as we vary $\epsilon$, window size, and step size. Workload error is a measure of variance. The square root of the variance (i.e., the standard deviation) is an upper bound on the expected absolute error. Therefore, the curves of the average absolute errors of the four schemes (Figure 6) are very similar to those about the workload errors (Figure 5).

We now evaluate the elapsed time. It consists of two parts: a) query planning time, i.e., the time for *DP* and *EMD* to plan how to answer queries (Sections 3 and 4), and b) query running time, i.e., the time of running the planned queries on streaming data. Let us first consider the query planning time. *DP* performs very efficiently. In all the cases, it takes less than 0.1 seconds. By contrast, *EMD* is less efficient; it spends around 18 seconds. However, it is still acceptable, since query planning can be done offline before queries start to run on the data streams. Now we report the running time. Figure 7 is the results of the four schemes. All the schemes are efficient. In particular, *DP* and *EMD* are equally efficient as *Base*. In all the cases, it takes *DP* and *EMD* less than 1 second to complete the processing. *Binary* has better performance, since each sliding window generally can be constructed by only a few nodes in the binary trees.

## 5.2 General Queries

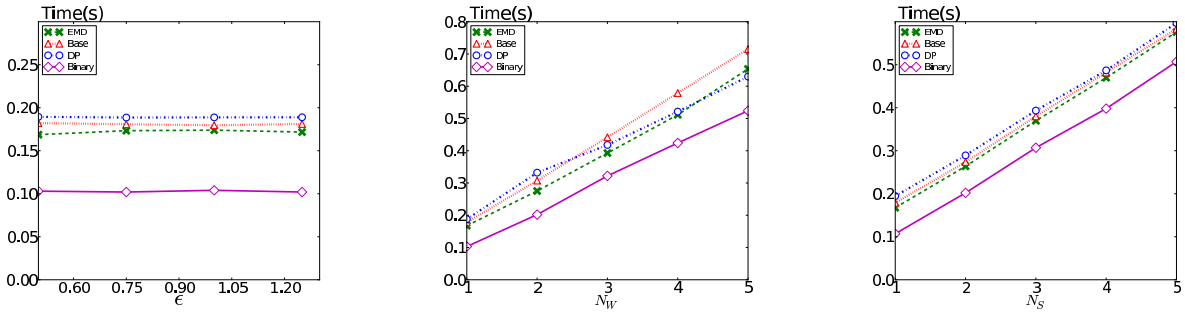
In general case, there is no restriction on the step size. We randomly select 10 steps from $[10, 100]$. Based on them, we also create 100 queries just like the experiments for the special queries. Since scheme *DP* is not applicable in the general case, we will only report the results of the other three methods.

As discussed in Section 2, the cycle length of a query processing system is the *Least Common Multiple* of the set of all the steps in the system. In general case, a step size can be an arbitrary positive integer, and the number of distinct steps can be big. Under such circumstances, the cycle length may be extremely big (can up to billions easily). In real-world applications each sliding window query is usually attached with an expiration time, which says how long the query can run before it expires. Let $ET$ be the smallest expiration time among all the submitted queries, and $LCM$ be the cycle length for the steps of the queries. We set $MEL = \min\{ET, LCM\}$. We process only queries that end before $MEL$. Queries beyond $MEL$ would not be processed, since they must be expired then. In the following, we set $ET$ to be $5,000$ by default.

Figure 8 and Figure 9 show the performance of the three schemes regarding workload error and absolute error, respectively. The trends of the curves in the two figures are similar to those in Figure 5 and Figure 6 for special queries. Once again, *EMD* clearly outperforms the two benchmarks, that is, *Base* and *Binary*.

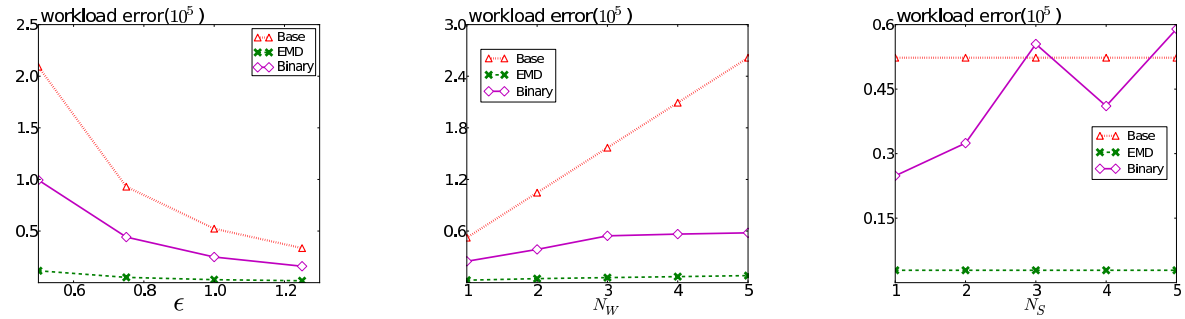
In order to further evaluate the performance of our approaches, we also examine the relative error. The experimental results of relative error for special queries are close to those for general queries. Due to space limitations, we only report the latter. We adopt three streams, that is, *uniform*, *adult*, and *census*, in the experiments. Figure 10 is the results when we vary $\epsilon$. When $\epsilon$ is larger, the privacy restriction becomes looser. So a lower magnitude of Laplace noise is added to the query answer, and the relative errors of all the schemes decrease.

Figure 11 investigates the effect of window size on the performance. Relative error is equal to $|approx - act|/act$, where $approx$ and $act$ are the approximate answer and actual answer for a query, respectively. For the simplicity of discussion, consider *uniform* stream (Figure 11 (a)), in which the numbers of 0's and 1's



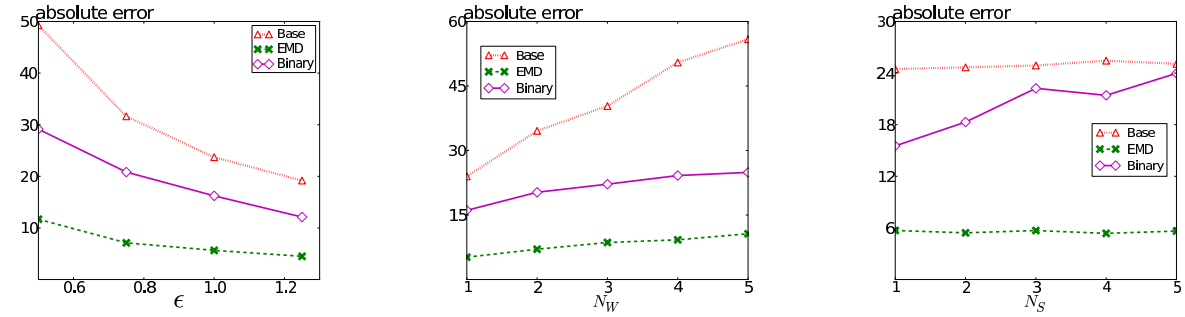

(a) varying $\epsilon$ (b) varying window size (c) varying step size

**Figure 7: Query running time (special case)**



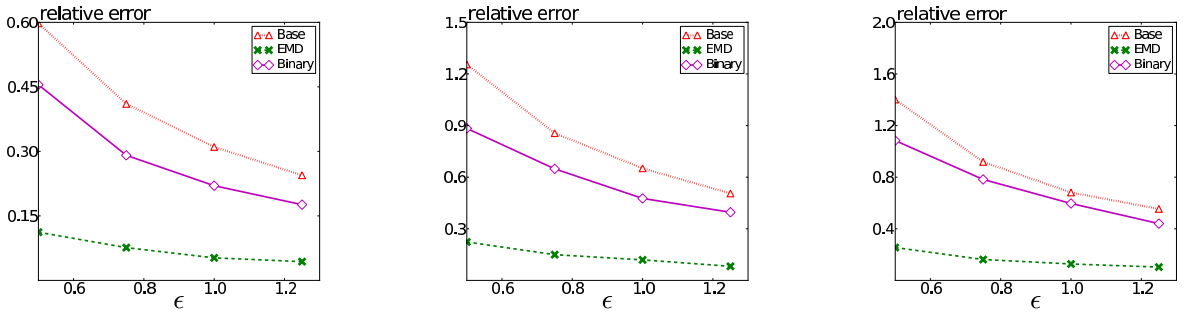

(a) varying $\epsilon$ (b) varying window size (b) varying step size

**Figure 8: Workload error (general case)**



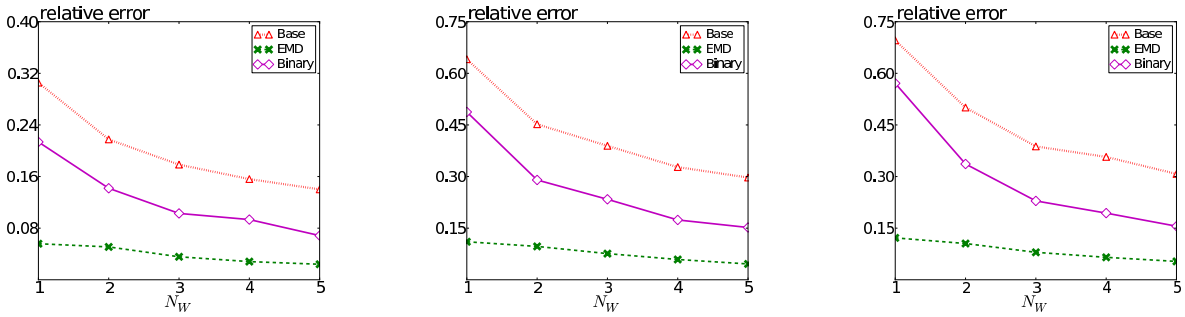

(a) varying $\epsilon$ (b) varying window size (b) varying step size

**Figure 9: Average absolute error (general case)**



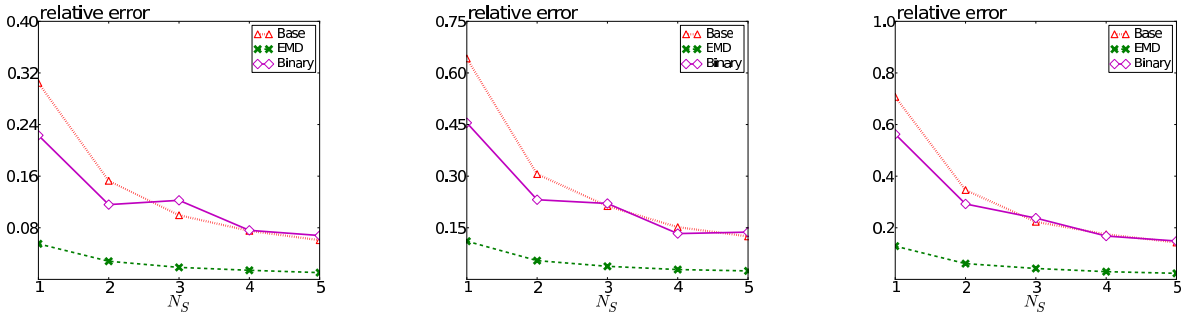

(a) uniform (b) adult (c) census

**Figure 10: Relative error by varying $\epsilon$**




(a) uniform (b) adult (c) census

**Figure 11: Relative error by varying window size**




(a) uniform (b) adult (c) census

**Figure 12: Relative error by varying step size**

in a window are roughly the same. As the window size increases by varying $N_W$ from 1 to 5, the value of $act$ (i.e., the number of 1's in a window) can be $N_W$ times as big as before. At the same time, the number of slots to compose a window could also be $N_W$ times as many as before. However, Laplace noise has a mean of 0; each independently generated noise can be equally likely to be positive and negative. Consequently, the noises (each for a slot composing the window) added to the window can partly cancel each other. Therefore, the numerator $|approx - act|$ (i.e., the summation of the noises) increases slower than the denominator $act$. Accordingly, when window size increases as a function of $N_W$, the relative error decreases. In Figure 12, relative error also decreases as step size becomes bigger. As the step size increases by varying $N_S$ from 1 to 5, the window size also increases. Thus, $act$ increases. However, the increase of step size does not change the number of slots to compose a window. So the noises (each for a slot) added to the window are almost the same (some variance can be incurred by the randomness of Laplace noise). Therefore, the relative error decreases as a function of $N_S$.

# 6. RELATED WORK

Privacy-preserving data release has been a very important research area for the past decade. A number of *syntactic* privacy models have been proposed, such as $k$-anonymity [29], $\ell$-diversity [25] and $t$-closeness [24]. However, these models have been shown to be vulnerable to attacks, and a new direction of *semantic* privacy has been pursued, culminating in the concept of *differential privacy* [12]. Differential privacy limits the disclosure risk due to the participation of an individual in a database, by adding random noise to all queries answered on the data. In recent years, numerous techniques [6, 19, 26, 20, 32, 22, 17, 31] have been proposed to improve the accuracy of differentially private queries under a variety of data sharing models and application scenarios. Related semantic models have also been investigated which focus on the increased risk to one's privacy due to participation in datasets [10, 15].

Differential privacy has a wide range of applications. In particular, Korolova et al. [20] develop an algorithm to privately publish queries and clicks for search logs. McSherry and Mironov [26] decompose popular recommendation algorithms in Netflix Prize competition into two phases: a learning phase guided by differential privacy, followed by a recommendation phase to provide personalized recommendation. These adaptive approaches yield accuracy comparable to their non-privacy-preserving counterparts. Xiao et al. [32] apply wavelet transforms on frequency matrix before adding noise. The resultant matrix can answer range-count queries with a higher accuracy than previous methods. Inan et al. [18] significantly improve the efficiency of private record matching [3] by a hybrid approach, which combines differential privacy with secure multi-party computation (SMC). Other applications of differential privacy include but not limited to releasing contingency tables [4], frequent pattern mining [5], computation of private coresets [16].

Recently, Li and Miklau [23] proposed a technique to answer a batch of queries. They first define a set of *strategy* queries, and then derive the answer for each query in the batch based on the strategy query results. However, their proposed algorithm and theoretical analysis are highly dependent on $L_2$ sensitivity for approximate differential privacy [26]. Whether the results are extendable to $L_1$ sensitivity is not clear. In addition, the technique is proposed for static settings. How to extend it to the context of data streams is unclear.

Closer to our work, differential privacy has been studied in the context of data streams [11, 13, 8, 14], where aggregate statistics

are continuously released as new data arrive. These approaches consider the model of a single-query on a binary stream, and release differentially private aggregate results about the number of 1's seen thus far in the stream. However, in practice users typically submit a large number of queries, and are only interested in a particular time window of the data, rather than the entire unbounded stream. Therefore, such techniques are insufficient in real-world applications. In contrast, we focus on how to release differentially private answers for a set of sliding window queries.

Rastogi and Nath [27] consider a recurring aggregate query on time-series data (e.g., a dataset which records the weights of participants in each month for a whole year). A recurring query can be seen as a sequence of $n$ queries $Q_1, Q_2, \ldots, Q_n$. (e.g., the 12 queries, each counting the number of participants with a weight larger than 100 KG in a distinct month). To improve the accuracy, they adopt Discrete Fourier Transform (DFT), and reduce the $n$ queries to $k$ ($k < n$) Fourier coefficients, which can approximately reconstruct the $n$ query answers. However, such an approach requires that the whole time-series dataset is available before the DFT is enforced. Consequently, it cannot be directly applied to usually unbounded data streams. Shi et al. [30] have also studied differential privacy on time-series data. They assume an untrusted aggregator, which sums the values periodically uploaded by a set of distributed users. To ensure the privacy of the users, the values are added noises before uploaded to the aggregator. Such an assumption is different from our scenario, in which the stream engine is trusted.

# 7. CONCLUSION AND FUTURE WORK

This paper proposed two solutions to release differentially private answers for a set of sliding window count queries. The first solution considers a special case, in which the size of a bigger step is always a multiple of that of a smaller one. The second one extends the research to the general case, in which the size of any step can be an arbitrary positive integer. The experimental evaluation demonstrated that our solutions are effective and efficient.

We have examined a data stream containing a single attribute. Still, a stream can have multiple ones. Hence, an interesting direction for the future work is to develop a scheme to release differentially private aggregate statistics of data streams with multiple attributes. An adversary may be able to observe the internal states of the solutions (i.e., algorithms to realize differential privacy). Therefore, another possible future work is to extend our solutions to achieve pan-privacy [14], which ensures differential privacy even if an adversary can observe the internal states of the solutions. In the experiments for both special and general cases, we tested 100 queries. It is also interesting to test the scalability of our solutions with respect to larger sets of queries.

# 8. ACKNOWLEDGMENTS

The work reported in this paper has been partially funded by NSF under grants CNS1016722 and CNS0964294. Qian Xiao and Kian-Lee Tan are partially supported by the grant R-252-000-433-305. The work of Gabriel Ghinita has been partially supported by the National Science Foundation under grant NSF-CNS 1111512. Ninghui Li's work was supported in part by the National Science Foundation under Grant No. 1116991.

# 9. REFERENCES

[1] http://archive.ics.uci.edu/ml/datasets/adult.

[2] http://www.ipums.org.

[3] R. Agrawal, A. V. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD*, pages 86–97, 2003.

[4] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *PODS*, pages 273–282, 2007.

[5] R. Bhaskar, S. Laxman, A. Smith, and A. Thakurta. Discovering frequent patterns in sensitive data. In *KDD*, pages 503–512, 2010.

[6] A. Blum, K. Ligett, and A. Roth. A learning theory approach to non-interactive database privacy. In *STOC*, pages 609–618, 2008.

[7] J. Bolot, N. Fawaz, S. Muthukrishnan, A. Nikolov, and N. Taft. Private sums on decayed streams. Technical report.

[8] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. In *ICALP*, volume 2, pages 405–417, 2010.

[9] G. Cormode, C. Procopiuc, E. Shen, D. Srivastava, and T. Yu. Differentially private spatial decompositions. In *To appear in ICDE*, 2012.

[10] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS*, pages 202–210, 2003.

[11] C. Dwork. Differential privacy in new settings. In *SODA*, pages 174–183, 2010.

[12] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[13] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *STOC*, pages 715–724, 2010.

[14] C. Dwork, M. Naor, T. Pitassi, G. N. Rothblum, and S. Yekhanin. Pan-private streaming algorithms. In *The First Symposium on Innovations in Computer Science*, pages 66–80, 2010.

[15] C. Dwork and K. Nissim. Privacy-preserving datamining on vertically partitioned databases. In *CRYPTO*, pages 528–544, 2004.

[16] D. Feldman, A. Fiat, H. Kaplan, and K. Nissim. Private coresets. In *STOC*, pages 361–370, 2009.

[17] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, 3(1):1021–1032, 2010.

[18] A. Inan, M. Kantarcioglu, G. Ghinita, and E. Bertino. Private record matching using differential privacy. In *EDBT*, pages 123–134, 2010.

[19] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What can we learn privately? In *FOCS*, pages 531–540, 2008.

[20] A. Korolova, K. Kenthapadi, N. Mishra, and A. Ntoulas. Releasing search queries and clicks privately. In *WWW*, pages 171–180, 2009.

[21] A. Levitin. *Introduction to the Design and Analysis of Algorithms (2nd Edition)*. Addison Wesley, 2007.

[22] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *PODS*, pages 123–134, 2010.

[23] C. Li and G. Miklau. An adaptive mechanism for accurate query answering under differential privacy. *PVLDB*, 5(6):514–525, 2012.

[24] N. Li, T. Li, and S. Venkatasubramanian. $t$-closeness: Privacy beyond $k$-anonymity and $\ell$-diversity. In *ICDE*, pages 106–115, 2007.

[25] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. $\ell$-diversity: Privacy beyond $k$-anonymity. In *ICDE*, number 24, 2006.

[26] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *KDD*, pages 627–636, 2009.

[27] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*, pages 735–746, 2010.

[28] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. In *International Journal of Computer Vision*, volume 40, pages 99–121, 2000.

[29] P. Samarati and L. Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *PODS*, page 188, 1998.

[30] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.

[31] X. Xiao, G. Bender, M. Hay, and J. Gehrke. ireduct: differential privacy with reduced relative errors. In *SIGMOD*, pages 229–40, 2011.

[32] X. Xiao, G. Wang, and J. Gehrke. Differential privacy via wavelet transforms. In *ICDE*, pages 225–236, 2010.