Privacy Integrated Data Stream Queries

Lucas Waye
Harvard University
lwaye@fas.harvard.edu

Abstract

Research on differential privacy is generally concerned with examining data sets that are static. Because the data sets do not change, every computation on them produces "one-shot" query results; the results do not change aside from randomness introduced for privacy . There are many circumstances, however, where this model does not apply, or is simply infeasible. Data streams are examples of non-static data sets where results may change as more data is streamed. Theoretical support for differential privacy with data streams has been researched in the form of differentially private streaming algorithms. In this paper, we present a practical framework for which a non-expert can perform differentially private operations on data streams. The system is built as an extension to PINQ (Privacy Integrated Queries), a differentially private programming framework for static data sets. The streaming extension provides a programmatic interface for the different types of streaming differential privacy from the literature so that the privacy trade-offs of each type of algorithm can be understood by a non-expert programmer.

Categories and Subject Descriptors H.3 [*Online Information Services*]: Data Sharing

Keywords Differential privacy; programming languages; privacy

1. Introduction

With the increase of big data services where personal user information is collected and stored in large quantities, the need for privacy is very important. *Differential privacy* [2] is a particularly strong definition of privacy that has withstood many known forms of privacy attacks. Differential privacy enforces that not much can be learned from a particular participant's data. To help make differential privacy more accessible, programming systems have been developed to help non-experts leverage the guarantees of differential privacy. Most of the research and techniques of differential privacy have been with respect to a single non-changing database. This precludes many environments where static data sets are not feasible. Situations where we would like to avoid performing queries on the entire data set include:

- The data is coming in over a long period of time and we would like to observe intermediate query results.
- It is practically infeasible to hold the entire data set in memory at one time.
- We would like to offer privacy guarantees against intrusions into the system running the computation (e.g. an intruder breaks into the system accessing the private database).

To address these issues, many have looked at differential privacy for streaming algorithms. Streaming algorithms can store less state than the entire database by summarizing relevant features of the data, produce intermediate outputs, and also in some cases protect against unannounced intrusions into the system. In this paper, we look at how differentially private streaming algorithms can be used by a non-expert in real-world applications where privacy is important.

1.1 Private Twitter

We are specifically motivated by scenarios where data is plentiful and generated continuously, but where privacy is still a concern. These scenarios come up in many places. For example in Dwork et al., a website that tracks H1N1 symptoms was described where the intention was to analyze aggregated information, possibly to track its growth or spread [3]. The privacy requirements of the web app were explored and potential privacy mechanisms were identified.

Consider another example web app with privacy expectations: a theoretical alternative to the popular social media website Twitter where Twitter messages (*tweets*) are visible to only a user's followers but where we would still like to analyze aggregate message patterns in a privacy-preserving way. In other words, a visitor can only see trends about tweets rather than specific tweets from specific users. Queries include examining topics that receive many tweets, perhaps specific to certain geographic areas. Other queries

PSP '14, October 21, 2014, Portland, OR, USA..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2296-6/14/10. . . $15.00.
http://dx.doi.org/10.1145/2687148.2687150

might include determining how many individuals are tweeting about certain topics. (Note these are different queries as the latter de-duplicates tweets from the same user.)

The need for privacy is apparent: users should be guaranteed that their tweets are only viewable to their followers so that they cannot be identified by outsiders, but the service may still like to compute aggregate information using their tweets. Differential Privacy is a good notion of privacy for this setting as any individual user or tweet cannot be singled out and identified.

Depending on the types of queries we are interested in, we find that we need to adjust our definitions of privacy to accommodate many different types of privacy, each with their own trade-offs:

- One may want their query results to be insensitive to correlated events. For example, a particular user frequently tweeting about a specific topic should not heavily influence the observed query results.
- One might also have different goals in mind for the trade-off between accuracy and responsiveness. That is, how frequently we get an intermediate result from the algorithm. One could do daily batches of the events or we may want updated query results every time an event occurs.
- Another important trade-off includes the amount of space the algorithm uses and somewhat related to that is how much privacy the algorithm preserves internally. In other words, how is a user's privacy affected if the algorithm's internal state is observed or leaked? For example, an algorithm that stores the last 100 events of a data stream could leak all information relating to those events if its internal state was observed.

These factors impact certain characteristics about an algorithm, including: privacy guarantees to each user, output behavior, and internal state of the algorithm. Related work provides theoretical results for these tradeoffs in the form of specialized algorithms that give various guarantees an analyst or data owner might desire. In order to be useful to a non-expert, the salient properties of each streaming algorithm should be clear in a unified framework so that the best algorithm can be chosen for the given constraints (e.g. privacy level or result accuracy) without regard to the algorithm's implementation.

1.2 Contributions

We extended the PINQ (Privacy Integrated Queries) platform [7] to support differentially private streaming algorithms[1]. *Streaming PINQ* provides a basis for handling streaming data, which was not readily available in PINQ or its underlying system for handling data (described later). The data analyst can program against Streaming PINQ in an intuitive way, similar to PINQ, but instead of receiving results directly after making a query, a handle to a streaming algorithm corresponding to the query is returned. The streaming algorithm has a common interface, but its behavior can vary depending the properties of the algorithm. We have implemented five different differentially private streaming algorithms. Our goal was not for a complete library based on the literature, but rather to get a good representative sample of different types of streaming algorithms that span the trade-offs of characteristics identified in this paper. We primarily focused on how the platform would handle the interaction of various types of streaming algorithms while maintaing an intuitive PINQ-like interface for the user of the platform.

[1] Code is available at http://git.io/jeWc2Q

2. Background

In this section we describe the definitions that will be used to reason about the privacy guarantees of the system. We will also describe the underlying platform, PINQ, that provides mechanisms for privately querying static data sets.

2.1 Differential Privacy

Differential privacy enforces that a resulting output distribution does not change much based on a particular individual's data. In particular, we use the following definition of differential privacy.

DEFINITION 2.1 (Differential Privacy). *A randomized algorithm* $\mathbf{M}$ *provides* ϵ*-differential privacy if for any two adjacent input data sets* $\mathbf{A}$ *and* $\mathbf{B}$*, and any set of possible outputs* $\mathbf{S}$ *of* $\mathbf{M}$*,*

$$\Pr[\mathbf{M}(\mathbf{A}) \in \mathbf{S}] \leq \mathbf{e}^{\epsilon} \Pr[\mathbf{M}(\mathbf{B}) \in \mathbf{S}]$$

Intuitively, this definition states that the outputs of differentially private mechanisms should not be *sensitive* to small changes in the input data sets. For example, a single change in an input data set should not affect the output of a differentially private mechanism very much. From a user's perspective, the mechanism's results will not change very much based on their participation in the data set. As a result, a user does not have to worry about their (possibly sensitive) information being identified based on the outputs of the mechanism. Other mechanisms, such as the removal of Personally Identifiable Information (e.g. name, social security number, etc.) do not have this property. Sweeney has shown that participants of a data set can be identified from looking at the output data set (original data set with Personally Identifiable Information removed) [11]. The framework presented in this paper is based on differentially privacy (as defined in Definition 2.1) as it is a particularly strong notion of privacy that is resistant to many known privacy attacks.

The sensitivity of a mechanism is based on *adjacency* of the inputs and outputs. In most settings where the input data sets are databases χ^n, adjacency would be databases where just one row is changed. This form of adjacency implies *row-level privacy* (i.e. just one row of a database is changed). Note that in a streaming setting, this may not capture an intuitive notion of privacy, so we will present different definitions of adjacency later.

An important concept in programming is the ability to easily compose operations together to form more complicated operations that can be reasoned about in a predictable way. Differential privacy is compatible with this concept. In particular, any sequential composition of differentially private computations implies differential privacy.

THEOREM 2.2 (Composition). *If $\mathbf{M_1}$ is an ϵ_1-differentially private algorithm and $\mathbf{M_2}$ is an ϵ_2-differentially private algorithm, then the sequential composition of $\mathbf{M_1}(\mathbf{X})$ followed by $\mathbf{M_2}(\mathbf{X})$ provides $(\epsilon_1 + \epsilon_2)$-differential privacy.*

A corollary of Theorem 2.2 is that a sequence of differentially private algorithms provides privacy equal to the sum of their ϵ values. Our programming framework makes use of composition to handle multiple streaming algorithms operating on the same data.

Another important theorem which we make use of involves operating over disjoint input sets. If differentially private algorithms are operating on disjoint subsets of the input data, then we can improve our guarantees.

THEOREM 2.3 (Disjointness). *If $\mathbf{M_1}$ and $\mathbf{M_2}$ are ϵ-differentially private algorithms and $\mathbf{X_1}$ and $\mathbf{X_2}$ are disjoint subsets of the input domain $\mathbf{X}$, then $\mathbf{M_1}(\mathbf{X_1})$ followed by $\mathbf{M_2}(\mathbf{X_2})$ provides ϵ-differential privacy.*

Intuitively, if algorithms are operating on separate data in parallel, then there is no further privacy lost. This theorem is useful when the input database contains many different properties that an analyst may want to explore independently without using up too much privacy from using composition.

2.2 Privacy Integrated Queries

```
var tweets = ReadAllSavedTweets("tweets.txt");
var agent = new PINQAgentBudget(1.0);
var data = new PINQueryable<Tweet>(tweets, agent);

double tweetsFromNY = data
  .Where(tweet => tweet.Location.State == "NY")
  .NoisyCount(1.0);

Console.WriteLine("Tweets from NY: " +
  tweetsFromNY);
```

Figure 1. PINQ program that provides a differentially private ($\epsilon = 1.0$) count of the number of tweets from New York.

There are a few differential privacy programming frameworks available. PINQ [7] is an extension to LINQ (Language Integrated Query) [8] which provides a programmer with a differentially private view of a given LINQ data source. In the basic feature set, it provides functionality to add Laplace random noise in accordance with an internal privacy budget on the data set that satisfies differential privacy. The privacy budget is enforced via an agent that is attached to the private query mechanism and is notified every time a differentially private operation takes place. The `PINQueryable` object is responsible for adding the noise and consulting the agent to check if the privacy budget is exceeded. As a result, when a developer wants to implement new differentially private primitive operations, it is the responsibility of the developer to prove its correctness. A user of the system must trust the underlying implementation of each `PINQueryable` implementation. The benefit is that it is very easy to extend the system, but introduces possible unsoundness in the system if a `PINQueryable` implementation is unsound. Other systems have a smaller trusted base and can enforce the privacy guarantees of new private functions (e.g. [9]).

Figure 1 shows an example PINQ program in the Private Twitter example. It loads all the tweets (stored in *"tweets.txt"*), then constructs a differentially private queryable object (named data) where $\epsilon = 1.0$. Then a query is performed on the data source which will yield a noisy version of the true number of tweets originating from New York. Note that the privacy budget of the data set in "tweets.txt" has been completely consumed, so no further queries should be performed on that data to preserve privacy.

2.3 Streaming Algorithms

In this paper, we are interested in streaming algorithms. The key difference between streaming algorithms and batch algorithms is that streaming algorithms do not have access to the data all at once. Instead, the data is coming through a stream one event at a time. Dwork et al. researched this setting and came up with various differentially private algorithms one could use with some extensions to the traditional definition of differential privacy [4].

Event-Level vs. User-Level Privacy An important change to support streaming differential privacy required re-defining the notion of adjacency for streams. Since a stream can be unbounded, the classic definition of row-level adjacency was not sufficient. Dwork et al. distinguish between two types of adjacency that give rise to two types of privacy: *event-level* privacy and *user-level* privacy. Intuitively, in event-level privacy one can think of adjacency between two streams as differing in just one event. In user-level privacy, the user could contribute to many events, so an adjacent stream would differ in all events a particular user contributed to. To formalize this notion of adjacency, we use the following definition.

DEFINITION 2.4 (X-Adjacent Data Streams). *Data streams (or stream prefixes) $\mathbf{S}$ and $\mathbf{S}'$ are X-adjacent if they differ only in the presence or absence of any number of occurrences of a single element $\mathbf{x} \in \mathbf{X}$. In other words, if all occurrences of $\mathbf{x}$ are deleted from both streams, then the resulting streams should be identical.*

Definition 2.4 gives rise to event-level and user-level privacy. In user-level privacy, users are distiguished by their values in the stream. That means that values in the stream need to correspond to a particular user (e.g. user row identifiers or unique names for each user).

Output Behavior Another important consideration with streaming algorithms is output behavior. Some streaming algorithms are single output algorithms [4]. That is, they can process a stream for an indefinite period of time, but once they receive a signal to output, they give one output and then need to be reinitialized. In a later publication, Dwork et al. provide a generalized transformation that can take a differentially private streaming algorithm that produces only one output to one that can produce continuous outputs [3]. It is important to note, though, that an upper bound on the number of events to process must be given as a parameter to the transformation upfront. Chan et al. present a differentially private counter with non-trivial error that does not require an upper bound on the number of events to process [1].

Pan-Privacy A nice characteristic of streaming algorithms is their ability to summarize large amounts of data in a size less than that of the stream. However, it is possible that the internal state of the algorithm may not be differentially private. Consider a private streaming counter that outputs noise over the true count while internally maintaining the true count. If an intrusion into the system were to occur, the algorithm would not maintain privacy as it leaked the true count. An algorithm that can maintain privacy under an intrusion is said to be *pan-private* [4]. It is also important to note how many intrusions the algorithm can withstand. Impossibility results are given for some finite-state algorithms (in particular, estimating user density in a stream) against more than one unannounced intrusion.

The above properties of streaming algorithms – which are in general not applicable to non-streaming algorithms – can be stated independently of each other. It is not clear which properties are advantageous over others; it is very dependent on the needs of the analyst and data owner. As a result, the Streaming PINQ framework handles each of these properties separately. For example, we provide a single output streaming algorithm as well as a continuous output streaming algorithm (at the expense of accuracy) so that the data analyst can pick the algorithm suitable to their particular needs. The only requirement that we make on algorithms is that they characterize their behaviors according to the above properties (i.e. event-level vs. user-level privacy, its output behavior, and its level of pan-privacy).

3. Streaming PINQ

Streaming PINQ is an extension to PINQ that supports streaming differentially private algorithms. It is implemented in roughly 1,000 lines of C# code. The framework is meant to have the same "look and feel" as PINQ. Many of the classes are entirely new and do not rely on the PINQ object model directly, but the same coding style is adopted for the programmer's ease of use and understanding.

To support streaming data, we designed a streaming data provider interface. The interface is extendable so that data providers can provide access to their own streams (e.g. network streams, file streams, etc.). To provide privacy, the data stream is wrapped around a `StreamingQueryable<T>` object that controls access to the events in the stream (in a similar fashion to `PINQueryable<T>` in PINQ). The wrapper object supports various transformations on the data. The underlying data cannot be accessed except through a differentially private streaming algorithm. The wrapper object consults an agent to determine whether it is safe to run a requested streaming algorithm. We extended the traditional `PINQAgent` class to distinguish between user-level and event-level private algorithms. PINQ did not have to distinguish between different notions of privacy that algorithms provided so it could have just one implementation to check access. When requesting a differentially private algorithm on the (possibly transformed) data, the `StreamingQueryable<T>` wrapper object returns a handle to the streaming algorithm for use by the programmer. The supported operations include: fetching the output of the algorithm, checking the number of events it has processed, and to start and stop it. Streaming algorithms are implemented on top of a base class `StreamingAlgorithm<T>`. Implementations of streaming algorithms must be trusted to be implemented correctly, as the platform requires that classes that extend `StreamingAlgorithm<T>` provide the privacy guarantees.

3.1 Streaming PINQ By Example

```
// The private source of all tweets
var tweetData = new AllTweetsStreamFireHose();

// Get differentially private view of tweetData
var tweets = new StreamingQueryable<Tweet>(
  tweetData, new UserLevelPrivacyBudget(1.0));

// Find users discussing #topic
var tweetedTag = tweets
  .Where(tweet => tweet.Message.Contains("#tag"))
  .Select(tweet => tweet.User)
  .UserDensityContinuous(0.5, AllUsers(), 10000);

// Every time a density is made, print it
tweetedTag.OnOutput = (d =>
  Console.WriteLine("Percent of users that  " +
    "tweeted #tag: " + d));

// Process 10000 events and stop
tweetedTag.ProcessEvents(10000);
```

Figure 2. example streaming PINQ program

Figure 2 shows an example streaming PINQ program for a setting similar to the one discussed in Section 1.1. The program will output an estimate of the fraction of users that have included "#tag" in their tweet every time a tweet is seen, and then stop outputting estimates after 10,000 tweets. Line 2 instantiates a streaming data provider. `tweets` wraps the private tweet data from the stream with the `UserLevelPrivacyBudget` agent (discussed in Section 3.3). Lines 9 to 11 transform the data to filter and select only users that are tweeting about a particular topic ("#tag"). Line 12 selects an algorithm that outputs an estimated user density (fraction of users who have appeared at least once in the stream) and makes an outputs at every event seen in the stream. The first parameter specifies the intended ϵ value to use, the second provides the algorithm with the data universe of users (a range of all possible users in the system), and the third parameter gives an upper bound on the number of events that can be processed. Line 15 assigns the output callback to a function which writes the current output to the console. Line 20 blocks the program and waits until 10,000 events have been processed, at which point the streaming algorithm stops listening for events. The next sections describe each component of the streaming PINQ implementation.

3.2 Streams

```
public abstract class StreamingDataSource<T>
{
  Action<T> EventReceived { get; set; }

  StreamingDataSource<T> filter(
    Func<T, bool> predicate);

  StreamingDataSource<U> map<U>(
    Func<T, U> mapper);
}
```

Figure 3. abstract base class for streaming data

The `StreamingDataSource<T>` class provides the basis for streaming data. It acts as a wrapper for a C# delegate with extra functionality provided for filtering and mapping the elements of the stream. Implementers of new data streams can extend this class and invoke the `EventReceived` delegate when a new event is received. Our implementation includes streaming data source implementations for reading from the console, generating random numbers, and also a functional data stream that has outputs dependent on the number of events sent.

The `filter` function behaves like an option monad with `null` values. In a stream setting, it is possible for events not to happen but a value should be given to the algorithm for that timestep. Dwork at al. referred to these types of events as the "nothing happened" element [3]. This element is encoded as `null`. So if a `null` event is received in the filter, it is immediately passed on to the target stream. Otherwise the filter predicate is run and if the filter matches then the event is passed to the target stream, otherwise `null` is passed. The same behavior is implemented for `map`, though the transformed data is passed to the target stream rather than the predicate check.[2]

3.3 Streaming Agents

```
public abstract class PINQStreamingAgent :
  PINQAgent
{
  bool ApplyEventLevel(double eps);
  double UnapplyEventLevel(double eps);
  bool ApplyUserLevel(double eps);
  double UnapplyUserLevel(double eps);
}
```

Figure 4. streaming agent that distinguishes between event-level and user-level privacy

Agents are responsible for enforcing that ϵ-differential privacy is preserved for the stream. Unlike `PINQAgent`, there are two notions of ϵ for streaming algorithms: one associated with event-level privacy and another associated with user-level privacy. Since the notion of user-level privacy is based on X-adjacency from Definition 2.4, we can see that event-level privacy is a special case of user-level privacy where the stream length is 1. As a result, if an ϵ-event level private algorithm runs for t time steps, it is $t\epsilon$-user level private. Also note that user-level privacy is a much stronger notion of event-level privacy. If an ϵ-user level private algorithm runs for t time steps, it is still just ϵ-event level private. Because there is a way to relate these algorithms, they can be used concurrently on the same stream. It should be noted, though, that user-level private algorithms have a much stronger definition of privacy so running an event-level private algorithm for even a short time has a very dramatic impact on the stream's user-level privacy.

The agent dynamically checks its budget every time an event is received. Since algorithms can *attach* (start receiving events from the stream) and *detach* (stop receiving events from the stream) from listening at any time, the agent is notified whenever an algorithm begins listening and when it stops listening. It is the responsibility of the agent to decide what to do when these events occur in order to preserve differential privacy. When an algorithm attaches to the stream, the agent is notified via `ApplyEventLevel` or `ApplyUserLevel` depending on the algorithm's privacy type it respects. The agent can return **true** allowing the algorithm to safely listen, or **false** if the algorithm should not be allowed to listen to events. When an algorithm detaches from the stream, the agent is notified via `UnapplyEventLevel` or `UnapplyUserLevel`. The agent will return how much privacy has been returned to the stream for future events.

From the `PINQStreamingAgent` base class, there are two inheriting classes that enforce either user-level privacy

[2] Note that the filter function is just a specialized map function. That is, `filter` can be implemented as `map(input => predicate(input) ? input : null)`

or event-level privacy. In the user-level privacy agent, privacy can never be returned to the stream, since the notion of adjacency extends to the entire stream, even after an algorithm stops listening. In other words, the algorithm has already learned something about the user, it can not "unlearn" it. On the other hand, when viewing the stream with event-level privacy, intuitively the agent only needs to make sure that at most ϵ is "learned" for each event. As a result, when an algorithm detaches it allows other algorithms to run afterwards with up to the entire initial privacy budget. The logic encapsulating these changes in ϵ is encoded in partially implemented abstract classes `PINQEventLevelAgent` and `PINQUserLevelAgent`.

We follow the same pattern as PINQ in establishing a budget that is defined by the data stream owner. The data stream owner provides the maximum amount of ϵ that can be used and also the type of privacy (event-level or user-level). Any algorithm or combination of algorithms that exceeds that level of ϵ is stopped. There are budget-based agents for both user-level privacy and event-level privacy named `PINQUserLevelAgentBudget` (which extends `PINQUserLevelAgent`) and `PINQEventLevelAgentBudget` (which extends `PINQEventLevelAgent`).

3.4 StreamingQueryable

The `StreamingQueryable<T>` class is the wrapper around a private stream in much the same way as `PINQueryable<T>` is in PINQ. It supports transformations on the data and instantiating streaming algorithms on the private data. It also keeps track of active streaming algorithm subscribers to the private stream data. The supported transformation operators are: `Where`, `Select`, and `Partition`. The first two transformations simply return a new `StreamingQueryable` with a filtered or mapped input data stream. The `Partition` transformation takes a set of possible keys and a key mapping function and produces a dictionary of keys mapping to their designated `StreamingQueryable` object. Figure 5 shows an example usage of the `Partition` transformation. Note that we make use of Theorem 2.3 to enforce that the amount of ϵ needed is the maximum ϵ used by the partitioned queryable objects. This ϵ is enforced in a similar way to PINQ through a specialized agent that tracks the used ϵ for each partitioned `StreamingQueryable`, as described in the previous section. In the case of Figure 5, an ϵ value of 1.0 on the original wrapper (`tweets`) would suffice.

A `StreamingQueryable` object tracks which algorithms are actively receiving events. It is the only part that directly receives events from the private stream (through the `EventReceived` delegate described in Section 3.2).

Every time an event is received, the `StreamingQueryable` object notifies its agent of all event-level algorithms that are attached and their corresponding ϵ values. If successful, the event is passed to every streaming algorithm for processing. After each algorithm has processed the event, the agent is notified that all event-level algorithms have detached. Note

```
var tweets = new StreamingQueryable<Tweet>(
  tweetData, new EventLevelPrivacyBudget(1.0));

var tweetsByState = tweets.Partition(
  AllStates(), tweet => tweet.Location.State);

var countsByState = new Dictionary<
  State, StreamingAlgorithm<State>>();

foreach (State state in tweetsByState.Keys) {
  countsByState[state] = tweetsByState[state]
    .BinaryCount(1.0, 10000);
  countsByState[state].StartListening();
}

Console.WriteLine("By State in last 10k events:")
foreach (State state in tweetsByState.Keys) {
  // Ensure 10,000 events have been processed
  countsByState[state].ProcessEvents(10000);
  Console.WriteLine(state + ": " +
    countsByState[state].LastOutput);
}
```

Figure 5. an example of partitioning disjoint events

that event level private algorithms are attached and detached (from the perspective of the agent) after every event so as to handle the case where user-level privacy is desired. Intuitively, this procedure simulates a user-level algorithm coming online at every time step, which is intuitively what is happening when an event-level private algorithm is used on a data stream that must respect user-level privacy.

For user-level private algorithms, the `StreamingQueryable` object only needs to notify the agent when they first register to receive events or when they finally unsubscribe from events on the stream. This book-keeping is handled in the `StreamingQueryable` object as it has direct access to the agent. It also frees streaming algorithms from needing to implement this logic. As a result, streaming algorithms can reason locally about their privacy and `StreamingQueryable` reasons about the overall privacy of many algorithms running on the same data stream.

3.5 Streaming Algorithms

```
public abstract class StreamingAlgorithm<T>
{
  Action<double> OnOutput { get; set; }
  int EventsSeen { get; protected set; }
  bool IsReceivingData { get; protected set; }

  void ProcessEvents(int n, bool stopAfterwards);
  abstract double GetOutput();
  double? LastOutput { get; protected set; }

  virtual void StartReceiving();
  virtual void StopReceiving();

  abstract void EventReceived(T data);
}
```

Figure 6. streaming algorithm base class

The `StreamingAlgorithm<T>` classes provide the mechanism for streaming differential privacy. The base class provides functionality to interact with the `StreamingQueryable` object to receive events. The base class also has functionality to get outputs made by the algorithm. This includes a convenience blocking mechanism that waits until a given number of events are processed. This method is implemented using a Semaphore. Subclasses generally only need to implement the `EventReceived` method to process the event. Algorithms differentiate between being user-level private and event-level private by extending the appropriate classes. That is, the type is used to differentiate between user-level private and event-level private algorithms.

We implemented five algorithms that span various characteristics, summarized in Figure 7:

1. Buffered Average batches the outputs it receives and then invokes PINQ's `NoisyAverage` on the buffered data when an output is requested.
2. Randomized Response Count will perform a randomized response on an event actually being seen so it holds no information about prior events on the stream. As a result, it has no private internal state so it is pan-private and works on an unbounded number of events.
3. Binary Counter maintains $\log T$ number of non-noisy partial sums (hence why it is not pan-private) and adds noise to each partial sum for output.
4. User Density creates a random sample of candidate users with their probability of being included in the count $\frac{1}{2}$. When a user that is in the random sample is seen, the probability of being included in the count is re-drawn with a probability $\frac{1}{2} + \frac{\epsilon}{4}$. The accuracy depends on how large the initial data universe sample is (size is computed in terms of α).
5. For the continuous bounded output case of User Density, the general transformation given in Dwork et al. was applied to User Density [3]. This transformation keeps a threshold of when to output a new result based on how much the original algorithm varies. If the algorithm varies frequently, accuracy is lost. For the User Density algorithm, the error after this transformation was calculated as being 6α.

For a more detailed description of the algorithms, please see the referenced papers in Figure 7.

3.6 Limitations and Future Work

This framework is designed for non-adversarial users. There are no formal checks on an implemented algorithm's advertised guarantees. For example, an implemented algorithm may be marked as pan-private but the implementation may be buggy and does not actually satisfy pan-privacy. Additionally, there are known attacks against PINQ [5] that Streaming PINQ is also susceptible to. For example, the `Where` method in `StreamingQueryable` accepts arbitrary C# predicates that can in general have side effects. As a result, an adversary could use a covert channel like timing to discover information about the private data (e.g. run a very long loop when encountering a row of interest).

In the cases where a data owner may prefer everyone to use only algorithms that provide a certain property (e.g. pan-privacy), one could imagine a scenario where a data analyst mistakenly uses an algorithm that does not have the data owner's desired properties. We leave it for future work to extend the system to enforce an algorithm's properties, much in the same way that user-level and event-level privacy is enforced. One could imagine a more granular agent that takes in the streaming algorithm object itself and compares that to a whitelist provided by the data owner, or (more ambitiously) even dynamically checks its code to assert its algorithm's advertised properties.

Another important caveat to streaming differentially private algorithms is that they expose the timing of when events are processed. Although implicit in the literature, one might want to hide the timing of events as it could reveal additional information about the event. For example, in analyzing a stock trading stream, a stock trade being made after normal trading hours may reveal additional information about the trade (e.g. that the trader is likely an institutional trader with special access to the exchange). These events could be mitigated by a time-boxed stream that buckets events into windows, or randomly dispersed into the stream if order was not important. This mechanism could be easily added to the streams described in Section 3.2, though it is unclear what the formal advantages are with this approach with respect to differential privacy.

This paper does not evaluate the performance and usefulness of the platform on actual data sets. Rather, it presents a design that extends a popular differential privacy programming framework with streaming in a modular way. We incorporated a few of the different notions of streaming differential privacy to show how new definitions of privacy can be added, but our aim was not to be exhaustive. For example, a generalization of event-level privacy called *w-event privacy* was not implemented in this framework [6]. We leave it as future work to evaluate this framework on real data sets. We hope that this work can serve as a practical benchmarking platform for experimenting with new streaming private algorithms and definitions.

4. Related Work

Fuzz, a programming system developed by Reed and Pierce, implements a language that guarantees differential privacy [9]. That is, any program written in Fuzz is differentially private. This makes it difficult, though, to introduce new differentially private primitives (such as dynamic data) into the system, as it would require modifying the language and compiler. It is not clear to the authors of this paper how Fuzz could be modified to support streaming data.

Algorithm	Privacy	Number of Outputs	Pan-Private	Additive Error
1. Buffered Average*	Event-Level	Single	No	$O(1)$
2. Randomized Response Count* [1]	Event-Level	Continuous Unbounded	Yes	$O(\sqrt{T})$
3. Binary Counter [1]	Event-Level	Continuous Bounded	No	$O((\log T)^{1.5})$
4. User Density [4]	User-Level	Single	Yes	α w.p. $1-\beta$
5. User Density Continuous [3, 4]	User-Level	Continuous Bounded	Yes	6α w.p. $1-\beta$

Figure 7. Implemented Streaming Algorithms. Note that ϵ is removed from accuracy measurements. α and β are user-defined parameters to the algorithm. Algorithms with an asterisk (*) denote known optimal accuracy for their listed properties. Buffered Average is simply adding just enough Laplace noise to achieve differential privacy. Randomized Response Count's error matches the theoretical lower bound from Dwork et al.'s negative result [3], given its properties (pan-private and continuous observation). Pan-Privacy is with respect to just one intrusion.

Airavat, another differentially private programming system, is a MapReduce based system that uses differentially private composition of computation [10]. The implementation of the system is based on Hadoop. This system is difficult to extend for streaming as Hadoop has been classically used for batch processing. Streaming data support in Hadoop is currently not well supported and would require much effort to incorporate into the Airavat system.

Streaming PINQ is built as an extension to PINQ. It only supports non-streaming data sets. This work extends it to support streaming data sets. We chose to extend PINQ to incorporate streaming as it allows easy extensibility, but places the proof burden of new mechanisms on the developer. PINQ is also developed as a library written in C#, a general purpose programming language, which allowed us to easily create a programmatic abstraction for streaming data that could be easily incorporated into the existing PINQ platform.

5. Conclusions

This paper describes an extension to PINQ that supports differentially private streaming algorithms. The platform allows a data analyst to choose the trade-offs in privacy vs. quality of the results. For example, a data analyst might want a very accurate result for user density, but would have to decrease the number of intermediate outputs seen. Also, if a data owner wants to enforce pan-privacy (since he or she may not trust the streaming algorithms to hold the data), then some algorithms would be unusable. We have built the platform to be flexible to allow the data owners and data analysts to decide which algorithms to use based on their needs without understanding the details of the algorithm's implementation. The only requirement the framework makes is that the algorithms provide a form of ϵ-differential privacy. (Whereas if privacy was not needed, then a far simpler framework could be used.)

We hope that the platform can serve as both a practical implementation for differentially private streaming algorithms that a data analyst could use, as well as provide a base for implementing and experimenting with new differentially private streaming algorithms.

Acknowledgments

We thank Salil Vadhan, Jonathan Ullman, and Stephen Chong for their comments on an earlier version of this paper and feedback on the project. We are grateful to the reviewers for their helpful comments.

References

[1] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3):26:1–26:24, Nov. 2011.

[2] C. Dwork. Differential privacy. In *ICALP*, pages 1–12. Springer, 2006.

[3] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proc. 42nd ACM symposium on Theory of computing*, STOC '10, pages 715–724, New York, NY, USA, 2010. ACM.

[4] C. Dwork, M. Naor, T. Pitassi, G. N. Rothblum, and S. Yekhanin. Pan-private streaming algorithms. In *Proc. ICS*, 2010.

[5] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. 20th USENIX conference on Security*, SEC'11, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[6] G. Kellaris, S. Papadopoulos, X. Xiao, and D. Papadias. Differentially private event sequences over infinite streams. *PVLDB*, 7(12):1155–1166, 2014.

[7] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, Sept. 2010.

[8] Microsoft. Linq (language integrated query).

[9] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. 15th ICFP*, ICFP '10, pages 157–168, New York, NY, USA, 2010. ACM.

[10] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for mapreduce. In *Proc. 7th NSDI*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[11] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *Journal of Law, Medicine & Ethics*, 25(2 & 3):98–110, 1997.