wPINQ: Differentially-Private Analysis of Weighted Datasets

Davide Proserpio
Boston University
dproserp@bu.edu

Sharon Goldberg
Boston University
goldbe@cs.bu.edu

Frank McSherry

ABSTRACT

We present an approach to differentially private computation in which one does not scale up the magnitude of noise for challenging queries, but rather scales *down* the contributions of challenging records. While scaling down all records uniformly is equivalent to scaling up the noise magnitude, we show that scaling records *non-uniformly* can result in substantially higher accuracy by bypassing the worst-case requirements of differential privacy for the noise magnitudes.

This paper details the data analysis platform **wPINQ**, which generalizes the Privacy Integrated Query (PINQ) to weighted datasets. [1]

1. INTRODUCTION

Differential Privacy (DP) has emerged as a standard for privacy-preserving data analysis. A number of platforms propose to lower the barrier to entry for analysts interested in differential privacy by presenting languages that guarantee that all written statements satisfy differential privacy [4, 5, 10, 11, 15]. However, these platforms have limited applicability to a broad set of analyses, in particular to the analysis of graphs, because they rely on DP's worst-case sensitivity bounds over multisets.

In this paper we present a platform for differentially private data analysis, wPINQ (for "weighted" PINQ), which uses *weighted datasets* to bypass many difficulties encountered when working with worst-case sensitivity. wPINQ follows the language-based approach of PINQ [10], offering a SQL-like declarative analysis language, but extends it to a broader class of datasets with more flexible operators, making it capable of graph analyses that PINQ, and other differential privacy platforms, are unable to perform. wPINQ also exploits a connection between DP and incremental computation to provide a random-walk-based probabilistic inference engine, capable of fitting synthetic datasets to arbitrary wPINQ measurements. For brevity this connection is dicussed in detail in the full version of the paper [13].

Compared to PINQ and other platforms, wPINQ is able to express and automatically confirm the privacy guarantees of a richer class of analyses, notably graph analyses, and automatically invoke inference techniques to synthesize representative datasets.

[1] An extended version of this paper appeared in VLDB15, and the full version inclding all the proofs is available at `http://arxiv.org/pdf/1203.3453.pdf`

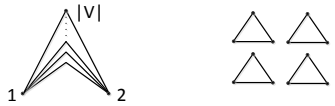

Figure 1: (left) Worst- and (right) best-case graphs for the problem of privately counting triangles.

1.1 Reducing sensitivity with weighted data

In the worst-case sensitivity framework, DP techniques protect privacy by giving noisy answers to queries against a dataset; the amplitude of the noise is determined by the sensitivity of the query. In a sensitive query, a single change in just one record of the input dataset can cause a large change in the query's answer, so substantial noise must be added to the answer to mask the presence or absence of that record. Because worst-case sensitivity is computed as the maximum change in the query's answer over *all possible input datasets*, this can result in significant noise, even if the input dataset is not worst-case. This problem is particularly pronounced in analysis of social graphs, where the presence or absence of a single edge can drastically change the result of a query [12, 14]. Consider the problem of counting triangles in the two graphs of Figure 1. The left graph has no triangles, but the addition of just one edge $(1, 2)$ immediately creates $|V| - 2$ triangles; to achieve differential privacy, the magnitude of the noise added to the query output is proportional to $|V| - 2$. This same amount of noise must be added to the output of the query on the right graph, even though it is not a "worst case" graph like the first graph.

We will bypass the need to add noise proportional to worst-case sensitivity by working with *weighted datasets*. Weighted datasets are a generalization of multisets, where records can appear an integer number of times, to sets in which records may appear with a real-valued multiplicity. Weighted datasets allows us to smoothly suppress individual 'troublesome' records (*i.e.,* records that necessitate extra noise to preserve privacy) by scaling down their influence in the output. We scale down the weight of these individual troublesome output records by the minimal amount they *do* require, rather than scale up the amplitude of the noise applied to all records by the maximal amount they *may* require. This *data-dependent rescaling* introduces inaccuracy only when and where the data call for it, bypassing many worst-case sensitivity bounds, especially for graphs.

Returning to the example in Figure 1, if the weight of each output triangle (a, b, c) is set to be $1/\max\{d_a, d_b, d_c\}$ where d_a is the degree of node a, the presence or absence of any single input edge can alter only a constant total

amount of weight across all output triangles. (This is because any edge (a, b) can create at most $\max\{d_a, d_b\}$ triangles.) Adding the weights of these triangles (the weighted analog of "counting") with constant-magnitude noise provides differential privacy, and can result in a more accurate measurement than counting the triangles with noise proportional to $|V|$ (which is equivalent to weighting all triangles with weight $1/|V|$). While this approach provides no improvement for the left graph, it can significantly improve accuracy for the right graph: since this graph has constant degree, triangles are measured with only constant noise.

It is worth noting how this approach differs from smooth sensitivity [12], which adds noise based on instance-dependent sensitivity; if the left and right graphs of Figure 1 were unioned into one graph, smooth sensitivity would still insist on a large amount of noise, whereas weighted datasets would allow the left half to be suppressed while the right half is not. This approach also differs from general linear queries allowing non-uniform weights [9]; these approaches apply only to linear queries (triangles is not) and require the non-uniformity to be explicitly specified in the query, rather than determined in a data-dependent manner. Weighted datasets are likely complementary to both of these other approaches.

In Section 2, we describe the design of wPINQ, a declarative programming language over weighted datasets, which generalizes PINQ [10]. While PINQ provided multiset transformations such as `Select`, `Where`, and `GroupBy`, the limitations of multisets meant that it lacked an effective implementation of the `Join` transformation, among others. wPINQ extends these to the case of weighted datasets, and uses data-dependent rescaling of record weights to enable a new, useful, `Join` operator, as well as several other useful transformations (*e.g.,* `Concat`, `SelectMany`, `Union`, *etc.*).

2. WEIGHTED DATASETS AND wPINQ

In order to design differentially-private algorithms that surmount worst-case sensitivity bounds by scaling down the influence of troublesome records, it will be convenient for us to work with *weighted datasets*. Section 2.1 discusses differential privacy (DP) for weighted datasets. The remainder of this section presents our design of **Weighted PINQ (wPINQ)**, a declarative programming language for weighted datasets that guarantees DP for all queries written in the language. The structure of wPINQ is very similar to its predecessor PINQ [10]; both languages apply a sequence of *stable transformations* to a dataset (Section 2.3), and then release results after a *differentially-private aggregation* (Section 2.2) is performed and the appropriate privacy costs are accumulated. Our main contribution in the design of wPINQ are new stable transformations operators (Figure 2) that leverage the flexibility of weighted datasets to rescale individual record weights in a data-dependent manner. We discuss these transformations in Sections 2.4–2.8.

2.1 Weighted Datasets & Differential Privacy

We can think of a traditional dataset (*i.e.,* a multiset) as function $A : D \to \mathbb{N}$ where $A(x)$ is non-negative integer representing the number of times record x appears in the dataset A. A weighted dataset extends the range of A to the real numbers, and corresponds to a function $A : D \to \mathbb{R}$ where $A(x)$ is the real-valued weight of record x. We define the difference between weighted datasets A and B as the sum of their element-wise differences:

$$\|A - B\| = \sum_x |A(x) - B(x)| \ .$$

We write $\|A\| = \sum_x |A(x)|$ for the size of a dataset.

In subsequent examples, we write datasets as sets of weighted records, where each is a pair of record and weight, omitting records with zero weight. To avoid confusion, we use real numbers with decimal points to represent weights. We use these two datasets in all our examples:

$$\begin{aligned} A &= \{(\text{“1”}, 0.75), (\text{“2”}, 2.0), (\text{“3”}, 1.0)\} \\ B &= \{(\text{“1”}, 3.0), (\text{“4”}, 2.0)\} \ . \end{aligned}$$

Here we have $A(\text{“2”}) = 2.0$ and $B(\text{“0”}) = 0.0$.

Differential privacy (DP) [3] generalizes to weighted datasets:

Definition 1. *A randomized computation M provides ϵ-*differential privacy *if for any weighted datasets A and B, and any set of possible outputs $S \subseteq Range(M)$,*

$$\Pr_M[M(A) \in S] \leq \Pr_M[M(B) \in S] \times \exp(\epsilon \times \|A - B\|) \ .$$

This definition is equivalent to the standard definition of differential privacy on datasets with non-negative integer weights, for which $\|A - B\|$ is equal to the symmetric difference between multisets A and B. It imposes additional constraints on datasets with non-integer weights. As in the standard definition, ϵ measures the *privacy cost* of the computation M (a smaller ϵ implies better privacy).

This definition satisfies sequential composition: A sequence of computations M_i each providing ϵ_i-DP is itself $\sum_i \epsilon_i$-DP. (This follows by rewriting the proofs of Theorem 3 in [10] to use $\|\cdot\|$ rather than symmetric difference.) Like PINQ, wPINQ uses this property to track the cumulative privacy cost of a sequence of queries and ensure that they remain below a privacy budget before performing a measurement.

Privacy guarantees for graphs. In all wPINQ algorithms for graphs presented here, the input sensitive dataset is `edges` is a collection of edges (a, b) each with weight 1.0, *i.e.,* it is equivalent to a traditional dataset. wPINQ will treat `edges` as a weighted dataset, and perform ϵ-DP computations on it (which may involve manipulation of records and their weights). This approach provides the standard notion of "edge differential privacy" where DP masks the presence or absence of individual edges (as in *e.g.,* [6,7,12,16]). Importantly, even though we provide a standard differential privacy guarantee, our use of weighted datasets allow us to exploit a richer set of transformations and DP algorithms.

Edge differential privacy does not directly provide privacy guarantees for vertices, for example "vertex differential privacy" [1, 2, 8], in which the presence or absence of entire vertices are masked. While non-uniformly weighting edges in the input may be a good first step towards vertex differential privacy, determining an appropriate weighting for the `edges` input is non-trivial and we will not investigate it here.

2.2 Differentially private aggregation

One of the most common DP mechanisms is the "noisy histogram", where disjoint subsets of the dataset are counted (forming a histogram) and independent values drawn from a Laplace distribution ("noise") are added to each count [3]. wPINQ supports this aggregation with the `NoisyCount`(A, ϵ) operator, which adds random noise from the Laplace$(1/\epsilon)$

`Select`	:	per-record transformation
`Where`	:	per-record filtering
`SelectMany`	:	per-record one-to-many transformation
`GroupBy`	:	groups inputs by key
`Shave`	:	breaks one weighted record into several
`Join`	:	matches pairs of inputs by key
`Union`	:	per-record maximum of weights
`Intersect`	:	per-record minimum of weights

Figure 2: Several stable transformations in wPINQ.

distribution (of mean zero and variance $2/\epsilon^2$) to the weight of each record x in the domain of A:

$$\texttt{NoisyCount}(A,\epsilon)(x) \quad = \quad A(x) + \text{Laplace}(1/\epsilon) \; .$$

$\texttt{NoisyCount}(A,\epsilon)$ provides ϵ-differential privacy, where the proof follows from the proof of [3], substituting $\|\cdot\|$ for symmetric difference. Importantly, we do *not* scale up the magnitude of the noise as a function of query sensitivity, as we will instead scale down the weights contributed by records to achieve the same goal.

To preserve differential privacy, `NoisyCount` must return a noisy value for every record x in the domain of A, even if x is not present in the dataset, *i.e.,* $A(x) = 0$. With weighted datasets, the domain of A can be arbitrary large. Thus, wPINQ implements `NoisyCount` with a dictionary mapping only those records with non-zero weight to a noisy count. If `NoisyCount` is asked for a record x with $A(x) = 0$, it returns fresh independent Laplace noise, which is then recorded and reproduced for later queries for the same record x.

Example. If we apply `NoisyCount` to the sample dataset A with noise parameter ϵ, the results for "0", "1", and "2" would be distributed as

$$\begin{aligned}
\texttt{NoisyCount}(A,\epsilon)(\text{“0”}) &\sim 0.00 + \text{Laplace}(1/\epsilon) \; , \\
\texttt{NoisyCount}(A,\epsilon)(\text{“1”}) &\sim 0.75 + \text{Laplace}(1/\epsilon) \; , \\
\texttt{NoisyCount}(A,\epsilon)(\text{“2”}) &\sim 2.00 + \text{Laplace}(1/\epsilon) \; .
\end{aligned}$$

The result for "0" would only be determined (and then recorded) when the value of "0" is requested by the user.

2.3 Stable transformations

wPINQ rarely uses the `NoisyCount` directly on a input weighted dataset, but rather on the output of a *stable transformation* of one weighted dataset to another, defined as:

Definition 2. *A transformation $T : \mathbb{R}^D \to \mathbb{R}^R$ is* stable *if for any two datasets A and A'*

$$\|T(A) - T(A')\| \quad \leq \quad \|A - A'\| \; .$$

A binary transformation $T : (\mathbb{R}^{D_1} \times \mathbb{R}^{D_2}) \to \mathbb{R}^R$ is stable *if for any datasets A, A' and B, B'*

$$\|T(A,B) - T(A',B')\| \quad \leq \quad \|A - A'\| + \|B - B'\|$$

(This definition generalizes Definition 2 in [10], again with $\|\cdot\|$ in place of symmetric difference.) The composition $T_1(T_2(\cdot))$ of stable transformations T_1, T_2 is also stable. Stable transformation are useful because they can composed with DP aggregations without compromising privacy, as shown by the following theorem (generalized from [10]):

Theorem 1. *If T is stable* unary *transformation and M is an ϵ-differentially private aggregation, then $M(T(\cdot))$ is also ϵ-differentially private.*

Importantly, transformations themselves do not provide differential privacy; rather, the output of a sequence of transformations is only released by wPINQ after a differentially-private aggregation (`NoisyCount`), and the appropriate privacy cost is debited from the dataset's privacy budget.

Stability for binary transformations (*e.g.,* `Join`) is more subtle, in that a differentially private aggregation of the output reveals information about both inputs. If a dataset A is used multiple times in a query (*e.g.,* as both inputs to a self-join), the aggregation reveals information about the dataset multiple times. Specifically, if dataset A is used k times in a query with an ϵ-differentially-private aggregation, the result is $k\epsilon$-differentially private for A. The number of times a dataset is used in a query can be seen statically from the query plan, and wPINQ can use similar techniques as in PINQ to accumulate the appropriate multiple of ϵ for each input dataset.

The rest of this section presents wPINQ's stable transformations, and discuss how they rescale record weights to achieve stability. We start with several transformations whose behavior is similar to their implementations in PINQ (and indeed, LINQ): `Select`, `Where`, `GroupBy`, `Union`, `Intersect`, `Concat`, and `Except`. We then discuss `Join`, whose implementation as a stable transformation is a significant departure from the standard relational operator. We also discuss `Shave`, a new operator that decomposes one record with large weight into many records with smaller weights.

2.4 Select, Where, and SelectMany

We start with two of the most fundamental database operators: `Select` and `Where`. `Select` applies a function $f : D \to R$ to each input record:

$$\texttt{Select}(A,f)(x) \quad = \quad \sum_{y:f(y)=x} A(y) \; .$$

Importantly, this produces output records weighted by the *accumulated weight* of all input records that mapped to them. `Where` applies a predicate $p : D \to \{0,1\}$ to each record and yields those records satisfying the predicate.

$$\texttt{Where}(A,p)(x) \quad = \quad p(x) \times A(x) \; .$$

One can verify that both `Select` and `Where` are stable.

Example. Applying `Where` with predicate $x^2 < 5$ to our sample dataset A in Section 2.1 gives $\{(\text{“1”}, 0.75), (\text{“2”}, 2.0)\}$. Applying the `Select` transformation with $f(x) = x \mod 2$ to A, we obtain the dataset $\{(\text{“0”}, 2.0), (\text{“1”}, 1.75)\}$; this follows because the "1" and "3" records in A are reduced to the same output record ("1") and so their weights accumulate.

We also mention the `SelectMany` operator, that generalizes `Select` and `Where`. (Its full description is in our technical report.) `SelectMany` is a unary operator, adapted from LINQ, which allows one-to-many record transformation: `SelectMany` maps each record to a list of elements, and then outputs the (flattened) collection of all elements from all the produced lists. To provide a stable `SelectMany` operator, wPINQ scales down the weight of output records by the number of records produced by the same input record; *e.g.,* an input mapped to n items would be transformed into

n records with weights scaled down by n. Section 2.8 uses `SelectMany` to transform `edges` into a dataset of nodes.

2.5 GroupBy

The `GroupBy` operator implements functionality similar to MapReduce: it takes a key selector function ("mapper") and a result selector function ("reducer"), and transforms a dataset first into a collection of groups, determined by the key selector, and then applies the result selector to each group, finally outputting pairs of key and result. This transformation is used to process groups of records, for example, collecting edges by their source vertex, so that the out-degree of the vertex can be determined. The traditional implementation of `GroupBy` (*e.g.,* in LINQ) is not stable, even on inputs with integral weights, because the presence or absence of a record in the input can cause one group in the output be replaced by a different group, corresponding to an addition and subtraction of an output record. This can be addressed by setting the output weight of each group to be *half* the weight of the input records. This sufficient for most of the wPINQ examples in this paper, which only group unit-weight records.

We defer the more complex description of `GroupBy`'s behavior on general weighted datasets to our technical report.

Node degree. `GroupBy` can be used to obtain node degree:

```
//from (a,b) compute (a, da)
var degrees = edges.GroupBy(e => e.src, l => l.Count());
```

`GroupBy` groups edges by their source nodes, and then counts the number of edges in each group — this count is exactly equal to the degree d_a of the source node a — and outputs the record $\langle a, d_a \rangle$; since all input records have unit weight, the weight of each output record is 0.5.

2.6 Union, Intersect, Concat, and Except

wPINQ provides transformations that provide similar semantics to the familiar Union, Intersect, Concat, and Except database operators. `Union` and `Intersect` have weighted interpretations as element-wise min and max:

$$\begin{aligned} \texttt{Union}(A,B)(x) &= \max(A(x), B(x)) \\ \texttt{Intersect}(A,B)(x) &= \min(A(x), B(x)) \end{aligned}$$

`Concat` and `Except` can be viewed as element-wise addition and subtraction:

$$\begin{aligned} \texttt{Concat}(A,B)(x) &= A(x) + B(x) \\ \texttt{Except}(A,B)(x) &= A(x) - B(x) \end{aligned}$$

Each of these four are stable binary transformations.

Example. Applying $\texttt{Concat}(A, B)$ with our sample datasets A and B we get

$$\{(\text{“1”}, 3.75), (\text{“2”}, 2.0), (\text{“3”}, 1.0), (\text{“4”}, 2.0)\} .$$

and taking $\texttt{Intersect}(A, B)$ we get $\{(\text{“1”}, 0.75)\}$.

2.7 The Join transformation.

The `Join` operator is an excellent case study in the value of using weighted datasets, and the workhorse of our graph analysis queries.

Before describing wPINQ's `Join`, we discuss why the familiar SQL join operator fails to provide stability. The standard SQL relational equi-join takes two input datasets and a key selection function for each, and produces all pairs whose keys match. The transformation is not stable, because a single record in A or B could match as many as $\|B\|$ or $\|A\|$ records, and its presence or absence would cause the transformation's output to change by as many records [14]. To deal with this, the `Join` in PINQ suppresses matches that are not unique matches, damaging the output to gain stability, and providing little utility for graph analysis.

The wPINQ `Join` operator takes two weighted datasets, two key selection functions, and a reduction function to apply to each pair of records with equal keys. To determine the output of `Join` on two weighted datasets A, B, let A_k and B_k be their restrictions to those records mapping to a key k under their key selection functions. The weight of each record output from a match under key k will be scaled down by a factor of $\|A_k\| + \|B_k\|$. For the identity reduction function we can write this as

$$\texttt{Join}(A, B) = \sum_k \frac{A_k \times B_k^T}{\|A_k\| + \|B_k\|} \tag{1}$$

where the outer product $A_k \times B_k^T$ is the weighted collection of elements (a, b) each with weight $A_k(a) \times B_k(b)$. The proof that this `Join` operator is stable appears in the Appendix.

Example. Consider applying `Join` to our example datasets A and B using "parity" as the join key. Records with even and odd parity are

$$\begin{aligned} A_0 &= \{(\text{“2”}, 2.0)\} \quad \text{and} \quad A_1 = \{(\text{“1”}, 0.5), (\text{“3”}, 1.0)\} \\ B_0 &= \{(\text{“4”}, 2.0)\} \quad \text{and} \quad B_1 = \{(\text{“1”}, 3.0)\} \end{aligned}$$

The norms are $\|A_0\| + \|B_0\| = 4.0$ and $\|A_1\| + \|B_1\| = 4.5$, and scaling the outer products by these sums gives

$$\begin{aligned} A_0 \times B_0^T \;/\; 4.0 &= \{(\text{“}\langle 2, 4\rangle\text{”}, 1.0)\} \\ A_1 \times B_1^T \;/\; 4.5 &= \{(\text{“}\langle 1, 1\rangle\text{”}, 0.\overline{3}), (\text{“}\langle 3, 1\rangle\text{”}, 0.\overline{6})\} \end{aligned}$$

The final result is the accumulation of these two sets,

$$\{(\text{“}\langle 2, 4\rangle\text{”}, 1.0), (\text{“}\langle 1, 1\rangle\text{”}, 0.\overline{3}), (\text{“}\langle 3, 1\rangle\text{”}, 0.\overline{6})\} .$$

Join and paths. Properties of paths in a graph are an essential part of many graph analyses. We use `Join` to compute the set of all paths of length-two in a graph, starting from the `edges` dataset of Section 2.1:

```
//given edges (a,b) and (b,c) form paths (a,b,c)
var paths = edges.Join(edges, x => x.dst, y => y.src,
                       (x,y) => new Path(x,y))
```

We join `edges` with itself, matching edges (a, b) and (b, c) using the key selectors corresponding to "destination" and "source" respectively. Per (1), the result is a collection of paths (a, b, c) through the graph, each with weight $\frac{1}{2d_b}$, where d_b is the degree of node b. Paths through high-degree nodes have smaller weight; this makes sense because each edge incident on such a vertex participates in many length two paths. At the same time, paths through low degree nodes maintain a non-trivial weight, so they can be more accurately represented in the output without harming their privacy. By diminishing the weight of each path proportionately, the influence of any one edge in the input is equally masked by a constant amount of noise in aggregation.

2.8 Shave

Finally, we introduce an transformation that decomposes records with large weight to be into multiple records with smaller weight. Such an operator is important in many analyses with non-uniform scaling of weight, to ensure that each output record has a common scale.

The `Shave` transformation allows us to break up a record x of weight $A(x)$ into multiple records $\langle x, i\rangle$ of smaller weights w_i that sum to $A(x)$. Specifically, `Shave` takes in a function from records to a sequence of real values $f(x) = \langle w_0, w_1, \ldots\rangle$. For each record x in the input A, `Shave` produces records $\langle x, 0\rangle$, $\langle x, 1\rangle$, ... for as many terms as $\sum_i w_i \leq A(x)$. The weight of output record $\langle x, i\rangle$ is therefore

$$\texttt{Shave}(A, f)(\langle x, i\rangle) = \max(0, \min(f(x)_i, A(x) - \sum_{j<i} f(x)_j)) \ .$$

Example. Applying `Shave` to our sample dataset A where we let $f(x) = \langle 1.0, 1.0, 1.0, ...\rangle \ \forall x$, we obtain the dataset

$$\{(\text{“}\langle 1, 0\rangle\text{”}, 0.75), (\text{“}\langle 2, 0\rangle\text{”}, 1.0), (\text{“}\langle 2, 1\rangle\text{”}, 1.0), (\text{“}\langle 3, 0\rangle\text{”}, 1.0), \} \ .$$

`Select` is `Shave`'s functional inverse; applying `Select` with $f(\langle x, i\rangle) = x$ that retains only the first element of the tuple, recovers the original dataset A with no reduction in weight.

Transforming edges to nodes. We now show how to use `SelectMany`, `Shave`, and `Where` to transform the `edges` dataset, where each record is a unit-weight edge, to a `nodes` dataset where each record is a node of weight 0.5.

```
var nodes = graph.SelectMany(e => new int[] { e.a, e.b })
                 .Shave(0.5)
                 .Where((i,x) => i == 0)
                 .Select((i,x) => x);
```

`SelectMany` transforms the dataset of edge records into a dataset of node records, *i.e.,* each unit-weight edge is transformed into two node records, each of weight 0.5. In wPINQ, the weights of identical records accumulate, so each node x of degree d_x has weight $\frac{d_x}{2}$. Next, `Shave` is used convert each node record x into a multiple records $\langle x, i\rangle$ for $i = 0, ..., d_x$, each with weight 0.5. `Where` keeps only the 0.5-weight record $\langle x, 0\rangle$, and `Select` converts record $\langle x, 0\rangle$ into x. The result is a dataset of nodes, each of weight 0.5. Note that it is not possible to produce a collection of nodes with unit weight, because each edge uniquely identifies two nodes; the presence or absence of one edge results in *two* records changed in the output, so a weight of 0.5 is the largest we could reasonably expect from a stable transformation.

3. CONCLUSIONS

We have presented a new approach to differentially-private data analysis; instead of scaling up noise added to all records in the dataset to its worst-case sensitivity bounds, we instead move to context of *weighted datasets* so that we can scale down the contribution of specific troublesome records. We outlined the implementation of **wPINQ**, an extension of the PINQ programming language to *weighted datasets*, and we described in detail stable transformation we have implemented. In particular we have added versions of `SelectMany` and `Join` capable of smoothly reducing the weight associated with inputs that produce multiple outputs. We also discussed `Shave`, a new operator that decomposes one record with large weight into many records with smaller weights. More details about our wwPINQ platform, including the connection with probabilistic inference are available in [13]. The wPINQ full implementation can be downloaded at `http://cs-people.bu.edu/dproserp/wPINQ.html`.

4. REFERENCES

[1] J. Blocki, A. Blum, A. Datta, and O. Sheffet. Differentially private data analysis of social networks via restricted sensitivity. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 87–96. ACM, 2013.

[2] S. Chen and S. Zhou. Recursive mechanism: towards node differential privacy and unrestricted joins. In *SIGMOD'13*, pages 653–664. ACM, 2013.

[3] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. 2006.

[4] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, Jan. 2013.

[5] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[6] M. Hay, C. Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. In *IEEE ICDM '09*, pages 169 –178, dec. 2009.

[7] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. In *Proc. VLDB'11*, pages 1146–1157, 2011.

[8] S. Kasiviswanathan, K. Nissim, S. Raskhodnikova, and A. Smith. Graph analysis with node-level differential privacy, 2012.

[9] C. Li and G. Miklau. An adaptive mechanism for accurate query answering under differential privacy. pages 514–525, 2012.

[10] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD '09*, pages 19–30, 2009.

[11] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. Gupt: privacy preserving data analysis made easy. In *Proceedings of the 2012 international conference on Management of Data*, pages 349–360. ACM, 2012.

[12] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *ACM STOC '07*, pages 75–84, 2007.

[13] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis. *Proceedings of the VLDB Endowment*, 7(8), 2014.

[14] V. Rastogi, M. Hay, G. Miklau, and D. Suciu. Relationship privacy: output perturbation for queries with joins. In *Proc. PODS '09*, pages 107–116, 2009.

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

[16] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, 2011.