Calibrating Data to Sensitivity in Private Data Analysis

A Platform for Differentially-Private Analysis of Weighted Datasets

Davide Proserpio
Boston University
dproserp@bu.edu

Sharon Goldberg
Boston University
goldbe@cs.bu.edu

Frank McSherry
Microsoft Research
mcsherry@microsoft.com

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. Using a few simple operators (including a non-uniformly scaling Join operator) wPINQ can reproduce (and improve) several recent results on graph analysis and introduce new generalizations (*e.g.,* counting triangles with given degrees). We also show how to integrate probabilistic inference techniques to synthesize datasets respecting more complicated (and less easily interpreted) measurements.

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 [5, 6, 15, 16, 22]. 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 [15], 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. 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.

In an earlier workshop paper [18], we sketched a proposed workflow for DP graph analysis and presented preliminary results showing how these techniques could be used to synthesize graphs that respect degree and joint-degree distributions. This paper is a full treatment of our data analysis platform, including the wPINQ programming language and its formal properties, as well as its incremental query processing engine for probabilistic inference. We discuss why wPINQ is well-suited to solve problems in social graph analysis, and present new and more sophisticated use cases related to computing the distribution of triangles and motifs in social graphs, results of independent interest.

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 [17, 19]. 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 individ-

Proceedings of the VLDB Endowment, Vol. 7, No. 8

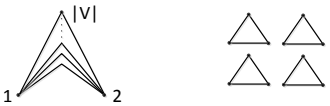

Figure 1: (left) Worst- and (right) best-case graphs for the problem of privately counting triangles.

ual '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 [17], 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 [13]; 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.

1.2 Our platform: wPINQ + MCMC

In Section 2, we describe the design of wPINQ, a declarative programming language over weighted datasets, which generalizes PINQ [15]. 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.*).

In Section 3 we show how wPINQ operators can implement a number of new graph analyses, focusing on subgraph-counting queries that informed the graph generator in [14]. We use a few lines of wPINQ to produce new algorithms to count triangles incident on vertices of degrees (d_1, d_2, d_3) each with noise proportional to $O(d_1^2 + d_2^2 + d_3^2)$, as well as length-4 cycles incident on vertices of degrees (d_1, d_2, d_3, d_4).

In Section 4 we show how wPINQ measurements can be analyzed with Markov Chain Monte Carlo (MCMC) techniques, to sample a synthetic dataset from the posterior distributions over datasets given wPINQ's noisy observations. This post-processing serves three purposes:

1. It can improve the accuracy of individual answers to wPINQ queries by removing obvious inconsistencies introduced by noise (*e.g.,* when the noisy count of triangles is a negative number, or not a multiple of six).
2. It can improve the accuracy of multiple measurements by combining the constraints they impose on each other (*e.g.,* the degree distribution and joint-degree distribution constrain each other; fitting a synthetic graph to both measurements produces more accurate results).
3. It can provide useful estimates for quantities that have not been directly queried in wPINQ, by evaluating them on the samples from the posterior distribution (*e.g.,* the joint-degree distribution constrains a graph's assortativity, *i.e.,* the extent to which nodes connect to other nodes with similar degrees, and the assortativity on the sampled graphs should be relatively accurate).

We detail the MCMC process, and the design and implementation of our efficient incremental re-execution platform that speeds up the iterative query re-evaluation at the heart of the MCMC process.

Finally, as a case study of the utility of our platform, Section 5 discusses the application of our platform to the problem counting triangles in a graph and evaluates its performance on several datasets.

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 [15]; 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:

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

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

Differential privacy (DP) [4] 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 [15] 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.,* [7, 9, 17, 23]). 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” [2, 3, 10], 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 [4]. wPINQ supports this aggregation with the $\texttt{NoisyCount}(A, \epsilon)$ operator, which adds random noise from the Laplace$(1/\epsilon)$ distribution (of mean zero and variance $2/\epsilon^2$) to the weight of each record x in the domain of A:

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

$\texttt{NoisyCount}(A, \epsilon)$ provides ϵ-differential privacy, where the proof follows from the proof of [4], 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.

`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.

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

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

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'*

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

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'*

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

(This definition generalizes Definition 2 in [15], 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 [15]):

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 [19]. 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

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

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

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

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

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

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 (Section 3). 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 $\sum_i w_i \le A(x)$. 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. GRAPH ANALYSES WITH wPINQ

In [18] we showed how the built-in operators in wPINQ can be used for differentially-private analysis of first- and second-order graph statistics; namely, we considered degree and joint-degree distributions, and reproduced the work of [7] and [23] as wPINQ algorithms while correcting minor issues in [7, 23]'s analyses (the requirement that the number of nodes be public and a flaw in the privacy analysis, respectively). These examples demonstrated that wPINQ is capable of expressing and bounding the privacy cost of challenging custom analyses.

We now present new results that use wPINQ to compute more sophisticated third- and fourth-order graph statistics. These algorithms suggest new approaches for counting triangles and squares in differentially-private manner (Theorem 2 and Theorem 3) which may be of independent interest.

3.1 Triangles by degree (TbD)

We now work through an example wPINQ analysis to count the number of triangles incident on vertices of degrees d_a, d_b, and d_c, for each triple $\langle d_a, d_b, d_c\rangle$. This statistic is useful for the graph generation model of [14]. The idea behind our algorithm is that if a triangle $\langle a, b, c\rangle$ exists in the graph, then paths *abc*, *cab*, and *bca* must exist as well. The algorithm forms each path *abc* and pairs it with the degree

of its internal node, d_b. The paths are then joined together, each time with a different rotation, so that *abc* matches *cab* and *bca*, identifying each triangle and supplying the degrees of each incident vertex.

The first action turns the undirected set of edges into a symmetric directed graph, by concatenating `edges` with its transpose.

```
var edges = edges.Select(x => new { x.dst, x.src })
                 .Concat(edges);
```

The implication of this transformation is that each subsequent use of `edges` will in fact correspond to two uses of the undirected source data.

We now compute length-two paths $\langle a, b, c\rangle$ by joining the set of edges with itself. We then use the `Where` transformation to discard length-two cycles (paths of the form $\langle a, b, a\rangle$).

```
var paths = edges.Join(edges, x => x.dst, y => y.src,
                       (x,y) => new Path(x, y))
                 .Where(p => p.a != p.c);
```

As in the example of Section 2.7 the weight of $\langle a, b, c\rangle$ is $\frac{1}{2d_b}$.

We next determine the degree of each vertex with a `GroupBy`, producing pairs $\langle v, d_v\rangle$ of vertex and degree, as in the example in Section 2.5. We join the result with the path triples $\langle a, b, c\rangle$, producing pairs of path and degree $\langle\langle a, b, c\rangle, d_b\rangle$.

```
var degs = edges.GroupBy(e => e.src, l => l.Count());
var abc = paths.Join(degs, abc => abc.b, d => d.key,
                     (abc,d) => new { abc, d.val });
```

The pairs `degs` produced by `GroupBy` each have weight 1/2, and joining them with `paths` results in records with weight

$$\frac{\frac{1}{2} \times \frac{1}{2d_b}}{\frac{1}{2} + d_b(d_b - 1) \times \frac{1}{2d_b}} = \frac{1}{2d_b^2} ,$$

per equation (1). The number of length-two paths through b is $d_b(d_b - 1)$ rather than d_b^2 because we discarded cycles.

We next use `Select` to create two rotations of $\langle\langle a, b, c\rangle, d_b\rangle$, namely $\langle\langle b, c, a\rangle, d_b\rangle$ and $\langle\langle c, a, b\rangle, d_b\rangle$. The resulting path-degree pairs hold the degrees of the first and third vertices on the path, respectively. Because they were produced with `Select`, their weights are unchanged.

```
var bca = abc.Select(x => { rotate(x.path), x.deg });
var cab = bca.Select(x => { rotate(x.path), x.deg });
```

We join each of the permutations using the length-two path as the key, retaining only the triple of degrees $\langle\langle d_a, d_b, d_c\rangle$.

```
var tris = abc.Join(bca, x => x.path, y => y.path,
                    (x,y) => new { x.path, x.deg, y.deg })
              .Join(cab, x => x.path, y => y.path,
                    (x,y) => new { y.deg, x.deg1, x.deg2 });
```

Each `Join` will be a unique match, since each path $\langle b, c, a\rangle$ occurs only once in the dataset. The weight of records in the output of the first `Join` is therefore

$$\frac{\frac{1}{2d_b^2} \times \frac{1}{2d_c^2}}{\frac{1}{2d_b^2} + \frac{1}{2d_c^2}} = \frac{1}{2d_b^2 + 2d_c^2} .$$

By the same reasoning, the weight of records in the output of the second `Join` (*i.e.,* the `tris` datasets), reflecting all three rotations and degrees, will be

$$\frac{1}{2(d_a^2 + d_b^2 + d_c^2)} . \qquad (2)$$

Finally, we sort each degree triple so that all six permutations of the degree triple $\langle d_a, d_b, d_c\rangle$ all result in the same record. We use `NoisyCount` to measure the total weight associated with each degree triple.

```
var order = triangles.Select(degrees => sort(degrees));
var output = order.NoisyCount(epsilon);
```

Each triangle $\langle a, b, c\rangle$ contributes weight $1/2(d_a^2 + d_b^2 + d_c^2)$ six times, increasing the weight of $\langle d_a, d_b, d_c\rangle$ by $3/(d_a^2+d_b^2+d_c^2)$. Nothing other than triangles contribute weight to any degree triple. Dividing each reported number by (2) yields the number of triangles with degrees d_a, d_b, d_c, plus noise.

This query uses the input dataset `edges` eighteen times (three permutations, each using `edges` three times, doubled due to the conversion (with `Concat`) to a symmetric directed edge set). To get ϵ-DP we must add Laplace noise with parameter $18/\epsilon$ to each count. Divided by $3/(d_a^2 + d_b^2 + d_c^2)$, the error associated with a triple $\langle d_a, d_b, d_c\rangle$ is a Laplace random variable with parameter $6(d_a^2 + d_b^2 + d_c^2)/\epsilon$. Thus,

THEOREM 2. *For an undirected graph G, let $\Delta(x, y, z)$ be the number of triangles in G incident on vertices of degree x, y, z. The mechanism that releases*

$$\Delta(x, y, z) + \mathit{Laplace}(6(x^2 + y^2 + z^2)/\epsilon)$$

for all x, y, z satisfies ϵ-differential privacy.

Section 5 discusses our experiments with this query.

3.2 Squares by degree (SbD)

Approaches similar to those used to count triangles can be used to design other subgraph-counting queries. To illustrate this, we present a new algorithm for counting squares (*i.e.,* cycles of length four) in a graph. The idea here is that if a square *abcd* exists in the graph, then paths *abcd*, *cadb* must exist too, so we find these paths and `Join` them together to discover a square.

The first three steps of the algorithm are identical to those in TbD; we again obtain the collection `abc` of length-two paths *abc* along with the degree d_b. Next, we join `abc` with itself, matching paths *abc* with paths *bcd*, to obtain length-three paths *abcd* with degrees d_b and d_c. We use the `Where` operator to discard cycles (*i.e.,* paths *abca*).

```
var abcd = abc.Join(abc, x => x.bc, y => y.ab,
                    (x, y) =>
                    { new Path(x.bc,y.ab), y.db, x.db })
              .Where(y => y.abcd.a != y.abcd.d);
```

We then use `Select` to rotate the paths in `abcd` twice:

```
var cdab = abcd.Select(x =>
            { rotate(rotate(x.abcd)), x.db, x.dc });
```

If a square *abcd* exists in the graph, then record $(abcd, d_d, d_a)$ will be in the rotated set `cdab`. We therefore join `abcd` with `cdab`, using the path as the joining key, to collect all four degrees d_a, d_b, d_c, d_d. Sorting the degrees coalesces the eight occurrences of each square (four rotations in each direction).

```
var squares = abcd.Join(cdab, x => x.abcd, y => y.abcd,
                (x, y) => new { y.da, x.db, x.dc, y.dd });

var order  = squares.Select(degrees => sort(degrees));
var output = order.NoisyCount(epsilon);
```

An analysis of the algorithm (deferred to our technical report) leads to the following new result:

THEOREM 3. *For an undirected graph G, let $\square(v,x,y,z)$ be the number of cycles of length 4 in G incident on vertices of degree v,x,y,z. The mechanism that releases*

$$\square(v,x,y,z) + \mathit{Laplace}(6(vx(v+x)+yz(y+z))/\epsilon)$$

for all v,x,y,z satisfies ϵ-differential privacy.

3.3 Counting arbitrary motifs

Motifs are small subgraphs, like triangles or squares, whose prevalence in graphs can indicate sociological phenomena. The approach we have taken, forming paths and then repeatedly `Join`ing them to tease out the appropriate graph structure, can be generalized to arbitrary connected subgraphs on k vertices. However, the analyses in this section were carefully constructed so that all records with the same degrees had exactly the same weight, allowing us to measure them separately and exactly interpret the meaning of each measurement. More general queries, including those for motifs, combine many records with varying weights, complicating the interpretation of the results. Fortunately, we introduce techniques to address this issue in the next section.

4. SYNTHESIZING INPUT DATASETS

DP queries can produce measurements that are not especially accurate, that exhibit inherent inconsistencies (due to noise), or that may not be directly useful for assessing other statistics of interest. One approach to these problems is the use of *probabilistic inference* [25], in which the precise probabilistic relationship between the secret dataset and the observed measurements is used, via Bayes' rule, to produce a posterior distribution over possible datasets. The posterior distribution integrates all available information about the secret dataset in a consistent form, and allows us to sample synthetic datasets on which we can evaluate arbitrary functions, even those without privacy guarantees.

While previous use of probabilistic inference (see *e.g.,* [25]) required human interpretation of the mathematics and custom implementation, wPINQ automatically converts all queries into an efficient Markov chain Monte Carlo (MCMC) based sampling algorithm. This automatic conversion is accomplished by an incremental data-parallel dataflow execution engine which, having computed and recorded the measurements required by the analyst, allows MCMC to efficiently explore the space of input datasets by repeatedly applying and evaluating small changes to synthetic inputs. Note that while this process does use the noisy wPINQ measurements, it no longer uses the secret input dataset; all synthetic datasets are public and guided only by their fit to the released differentially private wPINQ measurements.

4.1 Probabilistic Inference

The main property required for a principled use of probabilistic inference is an exact probabilistic relationship between the unknown input (*e.g.,* the secret graph) and the observed measurements (*e.g.,* the noisy answers to wPINQ queries). Although the input is unknown, we can draw inferences about it from the measurements if we know how likely each possible input is to produce an observed measurement.

For each query Q, dataset A, and measured observation m, there is a well-specified probability, $\Pr[Q(A)+\mathit{Noise}=m]$, that describes how likely each dataset A is to have produced an observation m under query Q. For example, when adding Laplace noise with parameter ϵ to multiple counts defined by $Q(A)$ (Section 2.2), that probability is

$$\Pr[Q(A)+\mathit{Noise}=m] \propto \exp(\epsilon \times \|Q(A)-m\|_1)\ . \quad (3)$$

While this probability is notationally more complicated when different values of ϵ are used for different parts of the query, its still easily evaluated.

This probability informs us about the relative likelihood that dataset A is in fact the unknown input dataset. Bayes' rule shows how these probabilities update a prior distribution to define a posterior distribution over datasets, conditioned on observations m:

$$\Pr[A|m] = \Pr[m|A] \times \frac{\Pr[A]}{\Pr[m]}\ .$$

$\Pr[m|A]$ is the probability m results from $Q(A)+N$, as in (3). $\Pr[m]$ is a normalizing term independent of A. $\Pr[A]$ reflects any prior distribution over datasets. The result is

$$\Pr[A|m] \propto \exp(\epsilon \times \|Q(A)-m\|_1) \times \Pr[A]\ .$$

The constant of proportionality does not depend on A, so we can use the right hand side to compare the relative probabilities $\Pr[A|m]$ and $\Pr[A'|m]$ for two datasets A and A'.

The posterior distribution focuses probability mass on datasets that most closely match the observed measurements, and indirectly on datasets matching other statistics that these measurements constrain. For example, while Theorem 2 indicates that DP measurement of the TbD requires significant noise to be added to high-degree tuples, the posterior distribution combines information from the TbD with highly accurate DP measurements of degree distribution [7, 18] to focus on graphs respecting both, effectively downplaying TdD measurements resulting primarily of noise and likely improving the fit to the total number of triangles.

However, the posterior distribution is a mathematical object, and we must still compute it, or an approximation to it, before we achieve its desireable properties.

4.2 Metropolis-Hastings

The posterior probability $\Pr[A|m]$ is over all datasets, an intractably large set. Rather than enumerate the probability for all datasets A, modern statistical approaches use sampling to draw representative datasets. Metropolis-Hastings is an MCMC algorithm that starts from a supplied prior over datasets, and combines a random walk over these datasets with a function scoring datasets, to produce a new random walk whose limiting distribution is proportional to the supplied scores. Eliding important (but satisfied) techincal assumptions, namely that the random walk be reversible and with a known statitionary distibution, the pseudo-code is:

```
var state;   // initial state

while (true)
{
   // random walk proposes new state
   var next = RandomWalk(state);

   // change state with probability Min(1, newScore/old)
   if (random.NextDouble() < Score(next) / Score(state))
      state = next;
}
```

The user provides an initial value for `state`, a random walk `RandomWalk`, and a scoring function `Score`.

Choosing the initial state. MCMC is meant to converge from any starting point, but it can converge to good answers faster from well-chosen starting points. Although we can start from a uniformly random dataset, we often seed the computation with a random dataset respecting some observed statistics. In the case of graphs, for example, we choose a random 'seed' graph matching wPINQ measurements of the secret graph's degree distribution (Section 5.1).

Choosing the random walk. We allow the user to specify the random walk, though a natural default is to replace a randomly chosen element of the collection with another element chosen at random from the domain of all possible input records. Our random walk for graphs is in Section 5.1. More advanced random walks exist, but elided assumptions (about reversibility, and easy computation of relative stationary probability) emerge as important details.

Choosing the scoring function. As we are interested in a distribution over inputs that better matches the observed measurements, we will take as scoring function

$$Score_{\langle Q,m\rangle}(A) = \exp(\epsilon \times \|Q(A) - m\|_1 \times pow) \ .$$

Our initial distribution over states is uniform, allowing us to to discard the prior distribution $\Pr[A]$ in the score function, and so when $pow = 1$ this score function results in a distribution proportional to $\exp(\epsilon \times \|Q(A) - m\|_1)$, proportional to the posterior distribution suggested by probabilistic inference. The parameter *pow* allows us to focus the output distribution, and make MCMC behave more like a greedy search for a single synthetic dataset. Large values of *pow* slow down the convergence of MCMC, but eventually result in outputs that more closely fit the measurements m.

4.3 Incremental query evaluator

In each iteration, MCMC must evaluate a scoring function on a small change to the candidate dataset. This scoring function essentially evaluates a wPINQ query on the changed data, which can be an expensive computation to perform once, and yet we will want to perform it repeatedly at high speeds. We therefore implement each wPINQ query as an *incrementally updateable computation*, using techniques from the view maintenance literature. Each MCMC iteration can thus proceed in the time it takes to incrementally update the computation in response to the proposed small change to the candidate dataset, rather than the time it takes to do the computation from scratch. We now describe this incremental implementation and its limitations.

To incrementally update a computation, one needs a description of how parts of the computation depend on the inputs, and each other. This is often done with a directed dataflow graph, in which each vertex corresponds to a transformation and each edge indicates the use of one transformation's output as another transformation's input. As an analyst frames queries, wPINQ records their transformations in such a directed dataflow graph. Once the query is evaluated on the first candidate dataset, a slight change to the input (as made in each iteration of MCMC) can propagate through the acyclic dataflow graph, until they ultimately arrive at the `NoisyCount` endpoints and incrementally update $\|Q(A) - m\|_1$ and the score function.

Each wPINQ transformation must be implemented to respond quickly to small changes it its input. Fortunately, all of wPINQ's transformations are *data-parallel*. A transformation is data-parallel if it can be described as a single transformation applied independently across a partitioning of its inputs. For example, `Join` is data-parallel because each of its inputs is first partitioned by key, and each part is then independently processed in an identical manner. Data-parallelism is at the heart of the stability for wPINQ's transformations (Section 2.3): a few changed input records only change the output of their associated parts. Data-parallelism also leads to efficient incremental implementations, where each transformation can maintain its inputs indexed by part, and only recomputes the output of parts that have changed. As these parts are typically very fine grained (*e.g.,* an individual `Join` key), very little work can be done to incrementally update transformations; outputs produced from keys whose inputs have not changed do not need to be reprocessed. All of wPINQ's transformations are data-parallel, and are either be implemented as a stateless pipeline operators (*e.g.,* `Select`, `Where`, `Concat`) or a stateful operators whose state is indexed by key (*e.g.,* `GroupBy`, `Intersect`, `Join`, `Shave`). The details of our incremental implementations of wPINQ's transformations are deferred to our technical report.

Two limitations to wPINQ's MCMC performance are computation time and memory capacity. Incrementally updating a computation takes time dependent on the number of weights that change; in complex graph analyses, a single changed edge can propagate into many changed intermediate results. At the same time, wPINQ maintains state in order to incrementally update a computation; this state also scales with the size of these intermediate results, which grow with the size of the input data size and with query complexity. In the TbD from Section 3.1, for example, one edge (a, b) participates in up to $O(d_a^2 + d_b^2)$ candidate triangles, each of which may need to have its weight updated when (a, b)'s weight changes. Also, the final `Join` operators in TbD match arbitrary length-two paths, which wPINQ stores indexed in memory; the memory required for the TbD therefore scales as $\sum_v d_v^2$ (the number of length-two paths) which can be much larger than number of edges $\sum_v d_v$.

Depending on the complexity of the query, wPINQ's MCMC iterations can complete in as few as 50 microseconds, though typical times are closer to hundreds of milliseconds on complex triangle queries. MCMC can require arbitrarily large amounts of memory as the size of the data grows; we have tested it on queries and datasets requiring as many as 64 gigabytes of memory. Distributed low-latency dataflow systems present an opportunity to scale wPINQ to clusters with larger aggregate memory. More detail on running time and memory footprint are reported in Section 5.3.

We plan to publicly release the implementation once we have properly cleaned and documented the code.

5. EXPERIMENTS

We now apply our wPINQ's query and probabilistic inference to the workflow for graph synthesis proposed in [18], extended with further measurements. We start by using wPINQ to produce noisy DP-measurements of a secret graph and then use MCMC to synthesize graphs respecting these noisy measurements. Our approach is similar to that of [14], who identify and measure key graph statistics which constrain broader graph properties. However, while [14] starts from *exact* degree correlation measurements, we start from

Graph	Nodes	Edges	$d_{\max}$	$\triangle$	r
CA-GrQc	5,242	28,980	81	48,260	0.66
CA-HepPh	12,008	237,010	491	3,358,499	0.63
CA-HepTh	9,877	51,971	65	28,339	0.27
Caltech	769	33,312	248	119,563	-0.06
Epinions	75,879	1,017,674	3,079	1,624,481	-0.01
Random(GrQc)	5,242	28,992	81	586	0.00
Random(HepPh)	11,996	237,190	504	323,867	0.04
Random(HepTh)	9,870	52,056	66	322	0.05
Random(Caltech)	771	33,368	238	50,269	0.17
Random(Epinion)	75,882	1,018,060	3,085	1,059,864	0.00

Table 1: Graph statistics: number of triangles (Δ), assortativity (r), and maximum degree $d_{\max}$.

DP measurements of degree correlation that can be noisy and inconsistent. In [18], we presented several positive results of applying our platform to this problem, generating synthetic graphs that respect degree distribution and/or joint degree distribution of a private graph. We do not reproduce these results here; instead, we present new results for the more challenging problem of counting triangles.

We start by presenting experiments on symmetric directed graphs with where the total privacy cost is a constant times ϵ where $\epsilon = 0.1$ and MCMC parameter $pow = 10,000$ (see Section 4.2). We investigate the sensitivity of our results to different values of ϵ in Section 5.3, and then close the loop on the discussion in Section 4.3 by experimenting with the scalability of our platform.

5.1 A workflow for graph synthesis (from [18])

To provide necessary background, we briefly reproduce the description of the workflow for graph synthesis that we sketched in [18]. The workflow begins with the analyst's wPINQ queries to the protected graph. Once the queries are executed and noisy measurements are obtained, the protected graph is discarded. Graph synthesis proceeds, using *only the noisy measurements*, as follows:

Phase 1. Create a "seed" synthetic graph. In [18] we showed wPINQ queries and regression techniques that result in a highly accurate ϵ-differentially private degree sequence. We then seed a simple graph generator that generates a random graph fitting the measured ϵ-DP degree sequence. This random graph is the initial state of our MCMC process.

Phase 2. MCMC. The synthetic graph is then fit to the wPINQ measurements, using MCMC to search for graphs that best fit the measurements. Starting from our seed graph, we use an edge-swapping random walk that preserves the degree distribution of the seed synthetic graph; at every iteration of MCMC, we propose replacing two random edges (a, b) and (c, d) with edges (a, d) and (c, b). As MCMC proceeds, the graph evolves to better fit the measurements.

5.2 Evaluating Triangles by Degree (TbD)

Our goal is now to combine MCMC with measurements of third-order degree correlations in order to draw inferences about graph properties we are not able to directly measure: specifically, assortativity r and the number of triangles $\triangle$. We find that it can be difficult to exactly reconstruct detailed statistics like the number of triangles with specified degrees, but that these measurements nonetheless provide information about aggregate quantities like r and $\triangle$.

We start by considering generating synthetic graphs using our triangles by degree (TbD) query described in Section 3.1. While this query has appealing bounds for small degrees, it still requires each $\langle d_a, d_b, d_c \rangle$ triple to have its

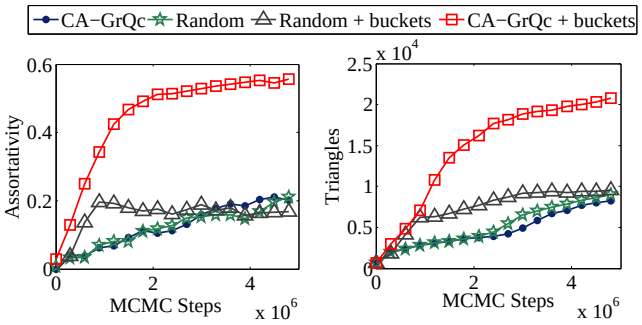

Figure 3: Behavior of the TbD query with and without bucketing for CA-GrQc. The values on CA-GrQc are $r = 0.66$, $\triangle = 48,260$ and on the sanity check $r = 0.0$, $\triangle = 586$. Bucketing substantially improves accuracy.

count perturbed by noise proportional to $6(d_a^2 + d_b^2 + d_c^2)/\epsilon$. For large degrees, many of which will not occur in the secret graph, the results are almost entirely noise. Unable to distinguish signal from noise, MCMC makes little progress.

Fortunately, the flexibility of our workflow allows for a simple remedy to this problem: instead of counting each degree triple individually, we first group triples into buckets with larger cumulative weight. The noise added to each bucket remains the same, but the weight (signal) in each bucket increases, which enables MCMC to distinguish graphs with different numbers of triangles at similar degrees.

The following modification to the code in Section 3.1 replaces each degree by the floor of the degree divided by k, effectively grouping each batch of k degrees into a bucket:

```
var degs = edges.GroupBy(e => e.src, l => l.Count()/k);
```

MCMC will automatically aim for synthetic graphs whose bucketed degree triples align with the measurements, without any further instruction about how the degrees relate.

Experiments. We experiment with our workflow using the graph CA-GrQc [11] (see statistics in Table 1). We first generate the 'seed' synthetic graph that is fit to wPINQ measurements of (a) degree sequence, (b) degree complementary cumulative density function, and (c) count of number of nodes (see [18] for details); the privacy cost of generating the seed synthetic graph is $3\epsilon = 0.3$. MCMC then fits the seed graph to TbD measurements with privacy cost $9\epsilon = 0.9$ for $\epsilon = 0.1$, so the total privacy cost of this analysis is $0.9 + 0.3 = 1.2$.

Figure 3 shows MCMC's progress (after 5×10^6 steps), plotting the number of triangles $\triangle$ and the assortativity r in the synthetic graph as MCMC proceeds. Figure 3 also plots MCMC's behavior when the secret graph is "Random(GrQc)", a random graph with the same degree distribution as CA-GrQc but with very few triangles, used to see if MCMC can distinguish the two. The figures reveal that MCMC is only able to distinguish the two graphs when bucketing is used ($k = 20$), but still does not find graphs respecting the input graph properties. This is likely due to a lack of signal in the result of the TbD query, which may be compounded by the approximate nature of MCMC or the restricted random walk we have chosen for it to use.

5.3 Evaluating Triangles by Intersect (TbI)

The TbD query of Section 3.1 has the property that the measurements are relatively easy to interpret, but this does

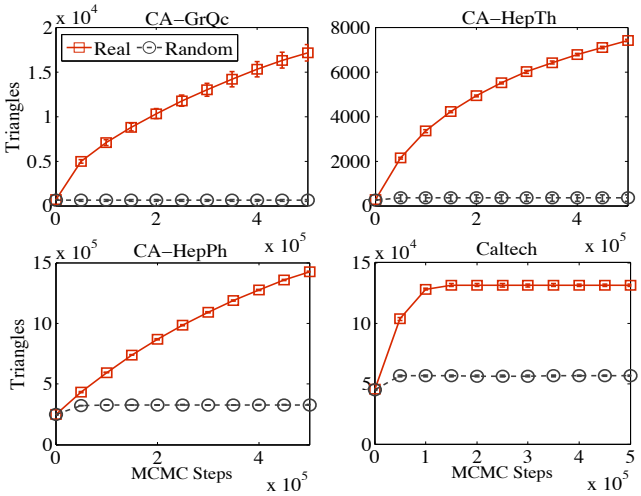

Figure 4: Fitting the number of triangles with TbI, 5×10^5 MCMC steps.

		CA-GrQc	CA-HepPh	CA-HepTh	Caltech
Δ	Seed	643	248,629	222	45,170
	MCMC	35,201	2,723,633	16,889	129,475
	Truth	48,260	3,358,499	28,339	119,563

Table 2: Δ before MCMC, after 5×10^6 MCMC steps using TbI, and in the original graph.

not neccesarily translate into good performance. We now consider an alternate query, whose results are harder to interpret directly, but that gives much better results when run through our MCMC workflow. The query, Triangles By Intersect (TbI) measures a single quantity related to the number of total triangles, reducing the amount of noise introduced. The algorithm is again based on the observation a triangle *abc* exists if and only if both of the length-two paths *abc* and *bca* exist. This time, however, we create the collection of length two paths, and *intersect* it with itself after appropriately permuting each path:

```
// form paths (abc) for a != c with weight 1/db
var paths = edges.Join(edges, x => x.dst, y => y.src,
                       (x,y) => new Path(x, y))
                 .Where(p => p.a != p.c);

//rotate paths and intersect for triangles
var triangles = paths.Select(x => rotate(x))
                     .Intersect(paths);

//count triangles all together
var result =  triangles.Select(y => "triangle!")
                       .NoisyCount(epsilon);
```

Deferring the analysis of weights to our technical report, we state the value TbI produces before wPINQ applies noise:

$$\sum_{\Delta(a,b,c)} \min\{\tfrac{1}{d_a}, \tfrac{1}{d_b}\} + \min\{\tfrac{1}{d_a}, \tfrac{1}{d_c}\} + \min\{\tfrac{1}{d_b}, \tfrac{1}{d_c}\} \qquad (4)$$

TbI uses the input dataset four times, incurring a privacy cost of 4ϵ, which is less than the 9ϵ incurred by TbD.

Notice that the TbI outputs only a single noised count, rather than counts by degree. While this single count might be difficult for a human to interpret, it contains information about triangles, and MCMC will work towards finding a synthetic graph that fits it.

Experiments. In Figure 4, we plot the number of triangles Δ versus the number of iterations of MCMC (5×10^5 steps) for synthetic graphs generated from our workflow, on both actual and random graphs. We generated the seed graphs as in Section 5.2 with privacy cost $3\epsilon = 0.3$. The TbI query has privacy cost $4\epsilon = 0.4$, and so the total privacy cost is $7\epsilon = 0.7$. We see a clear distinction between real graphs and random graphs; MCMC introduces triangles for the real graphs as appropriate, and does not for the random graphs. Table 2 reports the initial, final, and actual number of triangles for an order of magnitude more MCMC steps.

There is still room to improve the accuracy of triangle measurement using these techniques, but our results do show that even very simple measurements like TbI, which provide little direct information about the number of triangles, can combine with degree distribution information to give non-trivial insight into quantities that are not easily measured directly. wPINQ allows us to experiment with new queries, automatically incorporating the implications of new measurements without requiring new privacy analyses for each new query.

Different values of ϵ. We repeated the previous experiment with different values of $\epsilon \in \{0.01, 0.1, 1, 10\}$ (for total privacy cost 7ϵ). For brevity, we present results for the CA-GrQc graph and the corresponding random graph Random(GrQc) in Figure 5. We see that the choice of ϵ does not significantly impact the behavior of MCMC, maintaining roughly the same expected value but with increases in variance for larger values of ϵ (*i.e.,* noisier queries). MCMC remains well-behaved because the "signal" in the TbI query over the GrQC graph (*i.e.,* the value of equation (4)) is large enough to be distinguished from both random noise and the signal in a random graph.

Scalability analysis. We now experiment with TbI using larger datasets, to provide insight into the running time and memory requirements of our approach. The memory requirements should grow with $O(\sum_{v \in G} d_v^2)$, as these are the number of candidate triangles (Section 4.3). The running time should increase with the skew in the degree distribution, as each edge is incident on more candidate triangles.

To verify the scaling properties of TbI, we use five synthetic graphs drawn from the Barabàsi-Albert distribution. Barabàsi-Albert graphs follow a power law degree distribution (similar to some social networks) and the generation process uses a preferential attachment model. We fix the number of nodes at 100,000 and edges at $2M$ and change the degree of highest-degree nodes by increasing "dynamical exponent" of the preferential attachment [1] as $\beta \in \{0.5, 0.55, 0.6, 0.65, 0.7\}$. Experimental results are in Fig-

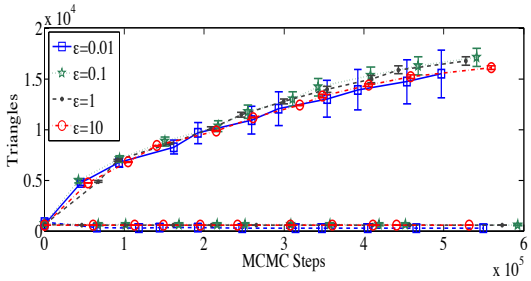

Figure 5: Testing TbI with different values of ϵ; error bars show standard deviation based on 5 repeated experiments. Total privacy cost for each query is 7ϵ. The position of the measurements on the x-axis have been randomized for better visualization.

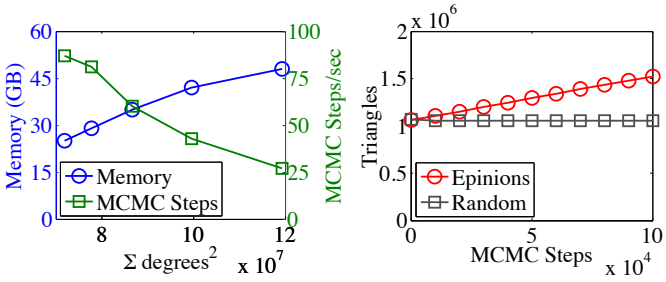

Figure 6: Running time and MCMC step/second for TbI computed on Barabàsi-Albert graphs with 100K nodes, 2M edges, and dynamical exponent of $\beta \in \{0.5, 0.55, 0.6, 0.65, 0.7\}$ (left). TbI behavior on the epinions graph (right).

ure 6. As $\sum_{v \in G} d_v^2$ increases from about 71M (for the graph with $\beta = 0.5$) to 119M (for $\beta = 0.7$), the memory required for MCMC increases. Meanwhile computation rate, *i.e.,* MCMC steps/second, decreases. Our platform can perform approximately 80 MCMC steps/second on the "easiest" ($\beta = 0.5$) graph using about 25Gb of RAM. For the most difficult graph ($\beta = 0.7$), it can perform about 25 MCMC steps/second using over 45Gb of RAM.

We conclude the scalability analysis with a performance evaluation of TbI on Epinions [21], see Table 1. While Epinions has about half the number of edges as the most difficult Barabàsi-Albert graphs we tried ($\beta = 0.7$), the quantity $\sum_{v \in G} d_v^2 = 224$M for Epinions is almost double that of the Barabàsi-Albert graph, making Epinions the most difficult graph we tried on our platform. To work with Epinions, we needed over 50GB of memory, and computation ran no faster than 10 MCMC steps/second. As usual, we experimented with this graph and a random graph with the same distribution but less triangles (statistics in Table 1). We run the MCMC for 100,000 steps and compute the number of triangles every 10,000 steps (Figure 6).

6. RELATED WORK

Bypassing worse case sensitivity. Since the introduction of differential privacy [4], there have been several approaches bypassing worst-case sensitivity [2,10,17,19]. In the smooth sensitivity framework [17], one adds noise determined from the sensitivity of the specific input dataset and those datasets near it. While these approaches can provide very accurate measurements, they typically require custom analyses and can still require large noise to be added even if only a few records in the dataset lead to worst-case sensitivity. Weighted datasets allow us to down-weight the specific records in the dataset that lead to high sensitivity, leaving the benign component relatively untouched.

New approaches [2,10] have surmounted sensitivity challenges by discarding portions of the input dataset that cause the sensitivity to be too high; for example, node-level differential privacy can be achieved by trimming the graph to one with a certain maximum degree and then using worst-case sensitivity bounds. Our approach can be seen as a smoother version of this; we scale down the weights of portions of the input dataset, instead of discarding them outright.

An alternate approach relaxes the privacy guarantee for the portions of the input dataset that cause sensitivity to be high. Investigating joins in social graphs, Rastogi *et al.* [19] consider a relaxation of differential privacy in which more information release is permitted for higher degree vertices. Our approach can be seen as making the opposite compromise, sacrificing accuracy guarantees for such records rather than privacy guarantees.

In [12,13] the authors use weighted sums with non-uniform weights to optimize collections of linear (summation) queries. While this is a popular class of queries, their techniques do not seem to apply to more general problems. With weighted datasets we can design more general transformations (*e.g.,* `Join`, `GroupBy`) that are crucial for graph analysis but not supported by [12,13].

Languages. Languages for differentially private computation started with PINQ [15], and have continued through Airavat [22], Fuzz [5,6], and GUPT [16]. To the best of our knowledge, none of these systems support data-dependent rescaling of record weights. Although Fuzz does draw a uniform rescaling operator (the "`!`" operator) from work of Reed and Pierce [20], the programmer is required to specify a *uniform* scaling constant for *all* records (and is essentially equivalent to scaling up noise magnitudes).

Privacy and graphs. Bespoke analyses for graph queries that provide edge-level differential privacy have recently emerged, including degree distributions [7], joint degree distribution (and assortativity) [23], triangle counting [17], generalizations of triangles [9], and clustering coefficient [24]. New results have emerged for node-level differential privacy as well [2,3,10]. Of course, any wPINQ analysis can be derived from first principles; our contribution over these approaches is not in enlarging the space of differentially private computation, but rather in automating proofs of privacy and the extraction of information.

[8] covers other graph analyses that satisfy privacy definitions that may not exhibit the robustness of DP.

Bibliographic note. As we mentioned throughout, our earlier workshop paper [18] sketched our workflow and presented preliminary results showing how it could be used to synthesize graphs that respect degree and joint-degree distributions. This paper is full treatment of our platform, showing how weighted transformation stability can form the basis of an expressive declarative programming language wPINQ, and presenting our incremental query processing engine for probabilistic inference. We also present new algorithms and experiments related to counting triangles and squares.

7. CONCLUSIONS

We have presented our platform for differentially-private computation that consists of a declarative programming language, wPINQ, and an incremental evaluation engine that enables MCMC methods to synthesize representative datasets. wPINQ is based on an approach to differentially-private computation, where data, rather than noise, is calibrated to the sensitivity of query. Specifically, wPINQ works with *weighted datasets* so that the contribution of specific troublesome records that can harm privacy (*e.g.,* edges incident on high-degree nodes) are smoothly scaled down. We have specialized our platform to private analysis of social graphs, and discussed how it can simplify the process, both by automating the proofs of privacy and the extraction of information needed for generating synthetic graphs. While we have cast a number of analyses as queries in wPINQ and evaluated their performance, the analyses we have shown

here are by no means the limit of what is possible with wPINQ. Indeed, we believe wPINQ's key benefit is its flexibility, and we therefore hope our platform will be an enabler for future private analyses of interesting datasets, especially social networks.

Acknowledgements. We thank the WOSN'12 reviewers, the VLDB'14 reviewers, George Kollios, Robert Lychev and Evimaria Terzi for comments on this draft.

8. REFERENCES

[1] A.-L. Barabási. Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987), 2013.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] M. Hay, K. Liu, G. Miklau, J. Pei, and E. Terzi. Tutorial on privacy-aware data management in information networks. Proc. SIGMOD'11, 2011.

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

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

[11] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

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

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

[14] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. In *SIGCOMM '06*, pages 135–146, 2006.

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

[16] 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.

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

[18] D. Proserpio, S. Goldberg, and F. McSherry. A workflow for differentially-private graph synthesis. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 13–18. ACM, 2012.

[19] 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.

[20] J. Reed and B. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. *ACM Sigplan Notices*, 45(9):157–168, 2010.

[21] M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In *The Semantic Web-ISWC 2003*, pages 351–368. Springer, 2003.

[22] 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.

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

[24] Y. Wang, X. Wu, J. Zhu, and Y. Xiang. On learning cluster coefficient of private networks. In *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*.

[25] O. Williams and F. McSherry. Probabilistic inference and differential privacy. *Proc. NIPS*, 2010.

APPENDIX

We prove the stability of `Join` discussed in Section 2.7.

THEOREM 4. *For any datasets A, A', B, B',*

$$\|\texttt{Join}(A,B) - \texttt{Join}(A',B')\| \quad \leq \quad \|A - A'\| + \|B - B'\| \,.$$

PROOF. We will first argue that

$$\|\texttt{Join}(A,B) - \texttt{Join}(A',B)\| \leq \|A - A'\| \,. \qquad (5)$$

An equivalent argument shows $\|\texttt{Join}(A',B) - \texttt{Join}(A',B')\| \leq \|B - B'\|$ and concludes the proof.

It suffices to prove (5) for any A_k, A'_k, B_k. Writing `Join` in vector notation as

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

we want to show that for each term in this sum,

$$\left\| \frac{A_k \times B_k^T}{\|A_k\| + \|B_k\|} - \frac{A'_k \times B_k^T}{\|A'_k\| + \|B_k\|} \right\| \leq \|A_k - A'_k\| \,.$$

The proof is essentially by cross-multiplication of the denominators and tasteful simplification. For simplicity, let a and b be $A_k \times B_k^T$ and $A'_k \times B_k^T$, respectively, and let x and y be the corresponding denominators. We apply the equality

$$\frac{a}{x} - \frac{b}{y} = \frac{a(y-x) - (b-a)x}{xy},$$

followed by the triangle inequality:

$$\|\frac{a}{x} - \frac{b}{y}\| \leq \frac{\|a(y-x)\|}{xy} + \frac{\|(b-a)x\|}{xy}.$$

Expanding, these two numerators can be re-written as

$$\begin{aligned} \|a(y-x)\| &= (\|A_k\| - \|A'_k\|) \times (\|A_k\|\|B_k\|) \\ &\leq \|A_k - A'_k\| \times (\|A_k\|\|B_k\|) \\ \|(b-a)x\| &= \|(A_k - A'_k)B_k\| \times (\|A_k\| + \|B_k\|) \\ &\leq \|A_k - A'_k\| \times \|B_k\|(\|A_k\| + \|B_k\|) \end{aligned}$$

Assuming, without loss of generality, that $\|A_k\| \geq \|A'_k\|$, their sum is at most

$$\begin{aligned} & \|A_k - A'_k\| \times (\|A_k\| + \|A'_k\| + \|B_k\|) \times \|B_k\| \\ = \; & \|A_k - A'_k\| \times (xy - \|A_k\|\|A'_k\|) \,. \end{aligned}$$

Division by xy results in at most $\|A_k - A'_k\|$. For our choice of a, b, x, y the term $\|b-a\|x$ has a factor of $|x-y|$ in it, which we extract from both terms. If we re-introduce the definitions of a, b, x, y and apply a substantial amount of simplification, this bound becomes

$$\|A_k - A'_k\| \times \frac{(\|A_k\| + \|A'_k\| + \|B_k\|) \times \|B_k\|}{(\|A_k\| + \|B_k\|) \times (\|A'_k\| + \|B_k\|)} \,.$$

The numerator is exactly $2\|A_k\|\|A'_k\|$ less than the denominator, making the fraction at most one. □