DOI:10.1145/1810891.1810916

Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis

By Frank McSherry

Abstract

Privacy Integrated Queries (PINQ) is an extensible data analysis platform designed to provide unconditional privacy guarantees for the records of the underlying data sets. PINQ provides analysts with access to records through an SQL-like declarative language (LINQ) amidst otherwise arbitrary C# code. At the same time, the design of PINQ's analysis language and its careful implementation provide formal guarantees of *differential privacy* for any and all uses of the platform. PINQ's guarantees require no trust placed in the expertise or diligence of the analysts, broadening the scope for design and deployment of privacy-preserving data analyses, especially by privacy nonexperts.

1. INTRODUCTION

Vast quantities of individual information are currently collected and analyzed by a broad spectrum of organizations. While these data clearly hold great potential for analysis, they are commonly collected under the premise of privacy. Careless disclosures can cause harm to the data's subjects and jeopardize future access to such sensitive information.

This has led to substantial interest in data analysis techniques with guarantees of privacy for the underlying records. Despite significant progress in the design of such algorithms, privacy results are subtle, numerous, and largely disparate. Myriad definitions, assumptions, and guarantees challenge even privacy experts to assess and adapt new techniques. Careful and diligent collaborations between nonexpert data analysts and data providers are all but impossible.

In an attempt to put much of the successful privacy research in the hands of privacy nonexperts, we designed the Privacy Integrated Queries (PINQ) language and runtime, in which all analyses are guaranteed to have one of the strongest unconditional privacy guarantees: *differential privacy*.[5, 8] Differential privacy requires that computations be formally indistinguishable when run with and without any one record, almost as if each participant had opted out of the data set. PINQ comprises a declarative programming language in which all written statements provide differential privacy, and an execution environment whose implementation respects the formal requirements of differential privacy.

Importantly, the privacy guarantees are provided by the platform itself; they require no privacy sophistication on the part of the platform's users. This is unlike much prior privacy research which often relies heavily on expert design and analysis to create analyses, and expert evaluation to vet proposed approaches. In such a mode, nonexpert analysts are unable to express themselves clearly or convincingly, and nonexpert providers are unable to verify or interpret their privacy guarantees. Here the platform itself serves as a common basis for trust, even for analysts and providers with no privacy expertise.

The advantage our approach has over prior platforms lies in differential privacy: its robust guarantees are compatible with many declarative operations and permit end-to-end analysis of arbitrary programs containing these operations. Its guarantees hold in the presence of arbitrary prior knowledge and for arbitrary subsequent behavior, simplifying the attack model and allowing realistic, incremental deployment. Its formal nature also enables unexpected new functionality, including grouping and joining records on sensitive attributes, the analysis of text and unstructured binary data, modular algorithm design (i.e., without whole-program knowledge), and analyses which integrate multiple data sources from distinct and mutually distrustful data providers.

The main restriction of this approach is that analysts can only operate on the data from a distance: the operations are restricted to declarative transformations and aggregations; no source or derived records are returned to the analysts. This restriction is not entirely unfamiliar to many analysts, who are unable to personally inspect large volumes of data. Instead, they write computer programs to distill the data to manageable aggregates, on which they base further analyses. While the proposed platform introduces a stricter boundary between analyst and data, it is not an entirely new one.

1.1. An overview of PINQ

We start by sketching the different aspects of PINQ that come together to provide a data analysis platform with differential privacy guarantees. Each of these sections are then developed further in the remaining sections of the note, but the high level descriptions here should give a taste for the different facets of the project.

Mathematics of PINQ: The mathematical basis of PINQ, differential privacy, requires any outcome of a computation over a set of records be almost as likely with and without any one of those records. Computations with this guarantee behave, from the point of view of each participant, as if their data were never used. It is currently one of the strongest of privacy guarantees. The simplest example of a differentially private computation is noisy counting: releasing of the number of records in a data set perturbed by symmetric exponen-

The original version of this paper is entitled "Privacy Integrated Queries" and was published in the *Proceedings of SIGMOD 2009*.

tial (Laplace) noise. Many other simple aggregations (e.g., Sum, Average, Median, among others) have similarly accurate randomized analogs.

To allow nonexpert analysts to produce new differentially private computations, we introduce the use of transformations, applied to data before differentially private aggregations, without weakening the differential privacy guarantees. For several relational transformations, a changed record in the input data set always results in relatively few changes in the output data set. A differentially private analysis applied to transformed data masks changes in the transformation's output, and consequently masks changes in its input as well. The composed transformation and analysis will provide differential privacy, with a formal guarantee depending on the quantitative form of "relatively few," which we must determine for each transformation. Such transformations can be composed arbitrarily, by nonexpert analysts, and combined with differentially private aggregations will serve as our query language.

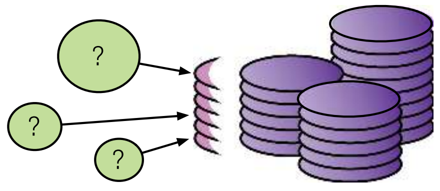
Finally, any sequence of differentially private computations also provides differential privacy; the quantitative privacy depletions are at worst additive (and occasionally better), and can be tracked online. Consequently, we can embed the query language above into any general purpose programming language, allowing arbitrary use of the results that return from the queries, as long as we monitor and constrain the total privacy depletion.

Implementation of PINQ: We have implemented a prototype of PINQ based on C#'s Language Integrated Queries (LINQ), an SQL-like declarative query language extension to .NET languages. Data providers use PINQ to wrap arbitrary LINQ data sources with a specified differential privacy allotment for each analyst. Analysts then write arbitrary C# programs, writing queries over PINQ data sources almost as if they were using unprotected LINQ data sources. PINQ's restricted language and run-time checks ensure that the provider's differential privacy requirements are respected, no matter how an analyst uses these protected data sets.

At a high level, PINQ allows the analyst to compose arbitrary queries over the source data, whose quantitative differential privacy guarantees are evaluated before the query is executed. If the analyst has framed a query whose privacy cost falls within the bounds prescribed by the data providers, the query is executed and the privacy cost subtracted from the amount available to the analyst for the associated data sets. If the cost falls outside the bounds, PINQ does not execute the query.

PINQ is designed as a thin layer in front of an existing query engine (Figure 1); it does not manage data or execute queries. Instead, it supplies differentially private implementations of common transformations and aggregations, themselves written in LINQ and executed by the LINQ providers of the underlying data sets. This approach substantially simplifies our implementation, but also allows a large degree of flexibility in its deployment. A data source only needs a LINQ interface to support PINQ, and we can take advantage of any investment and engineering put in to performant LINQ providers.

Figure 1. PINQ provides a thin protective layer in front of existing data sources, presenting an interface that appears to be that of the raw data itself.



We stress that PINQ represents a very modest code base; in its current implementation it is only 613 lines of C# code. The assessment logic, following the math, is uncomplicated. The aggregations must be carefully implemented to provide differential privacy, but these are most often only a matter of postprocessing the correct aggregate (e.g., adding noise). PINQ must also ensure that the submitted queries conform to our mathematical model for them. LINQ achieves substantial power by allowing general C# computations in predicates of `Where`, functions of `Select`, and other operations. PINQ must restrict and shepherd these computations to mitigate the potential for exploitation of side channels.

Applications of PINQ: Programming with PINQ is done through the declarative LINQ language, in an otherwise unconstrained C# program. The analyst is not given direct access to the underlying data; instead, information is extracted via PINQ's aggregation methods. In exchange for this indirection, the analyst's code is allowed to operate on unmasked, unaltered, live records.

With a few important exceptions, programs written with PINQ look almost identical to their counterparts in LINQ. The analysts assemble an arbitrary query from permitted transformations, and specify the accuracy for aggregations. Example 1 contains a C# PINQ fragment for counting distinct IP addresses issuing searches for an input query phrase.

Example 1. Counting searches from distinct users in PINQ.

We will develop this example into a more complex search log visualization application showcasing several of PINQ's

```
var data = new PINQueryable<SearchRecord>(...  ...);

var users = from record in data
            where record.Query == argv[0]
            groupby record.IPAddress;

Console.WriteLine(argv[0] + ":" + users.Count(0.1));
```

advantages over other approaches: rich data types, complex transformations, and integration into higher level applications, among many others. The full application is under 100 lines of code and took less than a day to write.

We have written several other examples of data analyses in PINQ, including k-means clustering, perceptron classification, and contingency table measurement. These examples have all been relatively easy adaptations of existing approaches.[2, 4]

2. MATHEMATICAL FOUNDATIONS

We now develop some supporting mathematics for PINQ. We review the privacy definition we use, differential privacy, and develop several properties necessary to design a programming language supporting its guarantees. Specifically, we discuss the data types we can support, common differentially private aggregations, how transformations of the data sets impact privacy, and how privacy guarantees of multiple analyses compose. All of our conclusions are immediate consequences of differential privacy, rather than additional assumptions or implementation details. The proofs are available in the full version of the paper.[10]

2.1. Differential privacy

Differential privacy is a relatively new privacy definition, building upon the work of Dwork et al.[8] and publicly articulated in Dwork.[5] It differs from most previous definitions in that it does not attempt to guarantee the prevention of data disclosures, privacy violations, or other bad events; instead, it guarantees that participation in the data set is not their cause.

The definition of differential privacy requires that a randomized computation yield nearly identical distributions over outcomes when executed on nearly identical input data sets. Treating the input data sets as multisets of records over an arbitrary domain and using $\ominus$ for symmetric difference (i.e., $A \ominus B$ is the set of records in A or B, but not both):

Definition 1. *A randomized computation M provides ϵ-* differential privacy *if for any two input data sets A and B, and any set of possible outputs S of M,*

$$\mathbf{Pr}[M(A) \in S] \quad \leq \quad \mathbf{Pr}[M(B) \in S] \times \exp(\epsilon \times |A \ominus B|).$$

For values of x much less than one, $\exp(x)$ is approximately $1 + x$. Differential privacy relies only on the assumption that the data sets are comprised of records, of any data type, and is most meaningful when there are few records for each participant, relative to $1/\epsilon$.

The definition is not difficult to motivate to nonexperts. A potential participant can choose between two inputs to the computation M: a data set containing their records (A) and the equivalent data set with their records removed (B). Their privacy concerns stem from the belief that these two inputs may lead to noticeably different outcomes for them. However, differential privacy requires that *any* output event (S) is almost as likely to occur with these records as without. From the point of view of any participant, computations which provide differential privacy behave almost as if their records had not been included in the analysis.

Taking a concrete example, consider the sensible concern of most Web search users that their name and search history might appear on the front page of the *New York Times*.[3] For each participant, there is some set S of outputs of M that would prompt the *New York Times* to this publication; we do not necessarily know what this set S of outputs is, but we need not define S for the privacy guarantees to hold. For all users, differential privacy ensures that the probability the *New York Times* publishes their name and search history is barely more than had it not been included as input to M. Unless the user has made the queries public in some other way, we imagine that this is improbable indeed.

2.2. Basic aggregations

The simplest differentially private aggregation (from Dwork et al.[8]) releases the number of records in a data set, after the addition of symmetric exponential (Laplace) noise, scaled by ϵ (Figure 2). The Laplace distribution is chosen because it has the property that the probability of any outcome decrease by a factor of $\exp(\epsilon)$ with each unit step away from its mean. Translating its mean (shifting the true value) by one unit scales the probability of any output by a multiplicative factor of at most $\exp(\epsilon)$. Changing an input data set from A to B can shift the true count by at most $|A \ominus B|$, and consequently a multiplicative change of at most $\exp(\epsilon \times |A \ominus B|)$ in the probability of any outcome.

Theorem 1. *The mechanism* $M(X) = |X| + $ *Laplace* $(1/\epsilon)$ *provides ϵ-differential privacy.*

The Laplace distribution has exponential tails in both directions, and the probability that the error exceeds t/ϵ in either direction is exponentially small in t. Consequently, the released counts are likely to be close to the true counts.

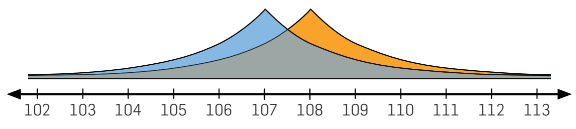
Other Primitive Aggregations: There are many other mechanisms that provide differential privacy; papers on the subject typically contain several. To date each has privacy established as above, by written mathematical proof based on intended behavior. While this is clearly an important step in developing such a computation, the guarantees are only as convincing as the proof is accessible and the implementation is correct.

Our goal is to enable the creation of as many differentially private computations as possible using only a few primitive components, whose mathematical properties and implementations can be publicly scrutinized and possibly verified. While we shouldn't preclude the introduction of novel primitives, they should be the exceptional, rather than default, approach to designing new differentially private algorithms.

2.3. Stable transformations

Rather than restrict programmers to a fixed set of aggregations, we intend to supply analysts with a programming language they can use to describe new and unforeseen computations. Most of the power of PINQ lies in arming the

Figure 2. Adding symmetric exponential noise to counts causes the probability of any output (or set of outputs) to increase or decrease by at most a multiplicative factor when the counts are translated.



analyst with a rich set of transformations to apply to the data set before differentially private aggregations.

Definition 2. *We say a transformation T is c-*stable *if for any two input data sets A and B,*

$$|T(A) \ominus T(B)| \;\leq\; c \times |A \ominus B|.$$

Transformations with bounded stability propagate differential privacy guarantees made of their outputs back to their inputs, scaled by their stability constant.

Theorem 2. *Let M provide ϵ-differential privacy, and let T be an arbitrary c-stable transformation. The composite computation $M \circ T$ provides $(\epsilon \times c)$-differential privacy.*

Once stability bounds are established for a set of transformations, a nonexpert analyst can combine any number of them as they see fit. Differential privacy bounds result from repeated application of Theorem 2, compounding the stability constants of the applied transformations with the ϵ value of the final aggregation.

Example Transformations: To give a sense for the types of stability bounds to expect, we consider a few representative transformations from LINQ.

The `Where` transformation takes an arbitrary predicate over records, and results in the subset of records satisfying the predicate. Any records in difference between A and B will result in at most those records in difference between their restrictions, resulting in a stability of one. Importantly, this is true independent of the supplied predicate; the predicate's logic can use arbitrarily sensitive information in the records and will still have stability one.

The `GroupBy` transformation takes a key selection function, mapping records to some key type, and results in a set of groups, one for each observed key, where each group contains the records mapped to the associated key value. For every record in difference between A and B, a group in the output can *change*. A change corresponds to symmetric difference two, not one; despite the apparent similarities in the groups, subsequent logic (e.g., a `Where`) can treat the two groups as arbitrarily different. As with `Where`, the stability of two holds for any key selection function, including those based on very sensitive fields or functions thereof.

The `Union` transformation takes a second data set, and results in the set of elements in either the first or the second data set. A record in difference between A and B results in no more than one record in difference in the output, yielding stability one. This is also true for records in difference in the second data set, giving us an example of a binary transformation. A differentially private analysis of the result of a binary transformation reflects information about both sources. This is uncomplicated unless the inputs derive from common data. Even so, a single change to a data set in common induces a bounded change in each of the transformation's inputs, and a bounded change in its output (i.e., the stability constants add).

The `Join` transformation takes a second data set, and key selection functions for both data sets. It intends to result in a set of pairs of records, one from each input, of records whose keys match. A single record in either set could match an unbounded number of records in the other set. Consequently, this important transformation has no stability bound. As we discuss later, there are restricted forms of `Join` that do have bounded stability (stability one, with respect to both inputs), but their semantics deviate from the unrestricted `Join` present in LINQ.

2.4. Composition

Any sequence of differentially private computations also provides differential privacy. Importantly, this is true even when subsequent computations can depend arbitrarily on the outcomes of the preceding computations.

Theorem 3. *Let M_i each provide ϵ_i-differential privacy. The sequence of $M_i(X)$ provides $(\Sigma_i \epsilon_i)$ -differential privacy.*

This simple theorem indicates that to track the cumulative privacy implications of several analyses, we need not do anything more complicated than add the privacy depletions.

If the queries are applied to disjoint subsets of the input domain we can improve the bound to the worst of the privacy guarantees, rather than the sum.

Theorem 4. *Let M_i each provide ϵ-differential privacy. Let D_i be arbitrary disjoint subsets of the input domain D. The sequence of $M_i(X \cap D_i)$ provides ϵ-differential privacy.*

Whereas sequential composition is critical for any functional privacy platform, parallel composition is a very important part of extracting good performance from a privacy platform. Although such operations could be analyzed as sequential composition, the privacy guarantee would scale with the number of subsets analyzed, often quite large.

2.5. A privacy calculus

The mathematics of this section allows us to quantitatively bound the privacy implications of arbitrary sequences of rich transformations and aggregations. This simplicity allows us to avoid burdening the analyst with the responsibility of correctly or completely describing the mathematical features of their query. Even for researchers familiar with the mathematics (e.g., the author), the reasoning process can be quite subtle and error-prone. Fortunately, it can be automated, the subject of Section 3.

3. IMPLEMENTING PINQ

PINQ is built atop C#'s LINQ. LINQ is a recent language extension to the .NET framework integrating declarative access to data streams (using a language very much like SQL) into arbitrary C# programs. Central to LINQ is the `IQueryable<T>` type, a generic sequence of records of type `T`. An `IQueryable` admits transformations such as `Where`, `GroupBy`, `Union`, `Join`, and more, returning new `IQueryable` objects over possibly new types. Only once an aggregation or enumeration is invoked is any computation performed; until this point the `IQueryable` only records the structure of the query and its data sources.

PINQ's implementation centers on a `PINQueryable<T>` generic type, wrapped around an underlying `IQueryable<T>`. This type supports the same methods as an `IQueryable`, but with implementations ensuring that the appropriate privacy calculations are conducted before any execution is invoked. Each `PINQueryable` is comprised of a private member data set (an `IQueryable`), and a second new data type, a `PINQAgent`, responsible for accepting or rejecting requested increments to epsilon. Aggregations test the associated `PINQAgent` to confirm that the increment to epsilon is acceptable before they execute. Transformations result in new `PINQueryable` objects with a transformed data source and a new `PINQAgent`, containing transformation-appropriate logic to forward modified epsilon requests to the agents of its source `PINQueryable` data sets.

The `PINQAgent` interface has one method, `Alert(epsilon)`, invoked before executing any differentially private aggregation with the appropriate value of epsilon, to confirm access. For `PINQueryable` objects wrapped around raw data sets, the `PINQAgent` is implemented by the data provider based on its privacy requirements, either from scratch or using one of several defaults (e.g., decrementing a per-analyst budget). For objects resulting from transformations of other `PINQueryable` data sets, PINQ constructs a `PINQAgent` which queries the `PINQAgent` objects of the transformation's inputs with transformation-appropriate scaled values of `epsilon`. These queries are be forwarded recursively, with appropriate values of epsilon, until all source data have been consulted. The process is sketched in Figure 3.

3.1. Aggregation operators

Each aggregation in PINQ takes `epsilon` as a parameter and provides ϵ-differential privacy with respect to its immediate data source. The privacy implications may be far worse for the underlying data sets from which this data set derives, and it is important to confirm the appropriately scaled amount of differentially private access. Before execution, each aggregation invokes the `Alert` method of their associated `PINQAgent` with this `epsilon`, conducting the aggregation only if the eventual response is positive.

`Count` is implemented as per Theorem 1, returning the accurate count of the underlying data plus Laplace noise whose magnitude is specified by the analyst, if large enough. Example 2 depicts the implementation of `Count`.

```
double Count(double epsilon)
{
    if (epsilon > 0.0 && myagent.Alert(epsilon))
        return mysource.Count() + Laplace(1.0/epsilon);
    else
        throw new Exception("Access is denied");
}
```

Example 2. [Abbreviated] Implementation of Count.

PINQ includes other aggregations—including `Sum`, `Average`, and `Median` among others—each of which takes epsilon and a function converting each record to a `double`. To provide differential privacy, the resulting values are first clamped to the interval [−1, +1] before they are aggregated. This is important to ensure that a single record has only a limited impact on the aggregate, allowing a relatively small perturbation to provide differential privacy.

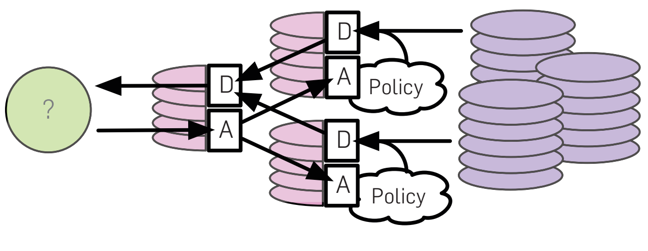
The implementations of these methods and the proofs of their privacy guarantees are largely prior work. `Sum`, like `Count`, is implemented via the addition of Laplace noise and is discussed in Dwork et al.[8] `Average` and `Median` are implemented using the exponential mechanism of McSherry and Talwar,[11] and output values in the range [−1, +1] with probabilities

$$\mathbf{Pr}[\texttt{Median}(A) = x] \propto \max_{\mathrm{med}(B)=x} \exp(-\epsilon \times |A \ominus B|/2)$$
$$\mathbf{Pr}[\texttt{Average}(A) = x] \propto \max_{\mathrm{avg}(B)=x} \exp(-\epsilon \times |A \ominus B|/2)$$

Each downweights the probability of a possible output x by (the exponentiation of) the fewest modifications to the input A needed to make x the correct answer.

The accuracy of `Average` is roughly $2/\epsilon$ divided by the number of records in the data set. `Median` results in a value which partitions the input records into two sets whose sizes differ by roughly an additive $2/\epsilon$; it need not be numerically close to the actual median.

Figure 3. PINQ control/data flow. An analyst initiates a request to a PINQ object, whose agent (A) confirms, recursively, differentially private access. Once approved by the providers' agents, data (D) flows back through trusted code ensuring the appropriate level of differential privacy.



3.2. Transformation operators

PINQ's flexibility derives from its transformation operators, each of which results in a new `PINQueryable` wrapped around an updated data source. The associated `PINQAgent` is wired to forward requests on to the participating source data sets before accepting, scaling `epsilon` by the transformation's stability constant.

Our implementations of many transformations are mostly a matter of constructing new `PINQueryable` and `PINQAgent` objects with the appropriate parameters. Some care is taken to restrict computations, as discussed in Section 3.4. Example 3 depicts the implementation of PINQ's `GroupBy`. Most transformations require similarly simple privacy logic.

EXAMPLE 3. [ABBREVIATED] IMPLEMENTATION OF GROUPBY.

```
PINQueryable<IGrouping<K,T>>
GroupBy<T,K>(Expression<Func<T,K>> keyFunc)
{
    // Section 3.4 explains this, and why it is needed
    keyFunc = Purify(keyFunc) as Expression<Func<T,K>>;

    // new agent with appropriate ancestor and stability
    var newagent = new PINQAgentUnary(this.agent, 2.0);

    // new data source reflecting the operation
    var newsource = this.source.GroupBy(keyFunc);

    // construct and return a new source and agent pair
    return   new   PINQueryable<IGrouping<K,T>>(newsource,
                                                 newagent);
}
```

The `Join` transformation is our main deviation from LINQ. To ensure stability one with respect to each input, we only report pairs that are the result of unique key matches. To ensure these semantics, PINQ's `Join` invokes LINQ's `GroupBy` on each input, using their key selection functions. Groups with more than one element are discarded, and the resulting singleton elements are joined using LINQ's `Join`.

While this clearly (and intentionally) interferes with standard uses of `Join`, analysts can reproduce its standard behavior by first invoking `GroupBy` on each data set, ensuring that there is at most one record per group, before invoking `Join`. The difference is that the `Join` is now required to reduce pairs of groups, rather than pairs of records. Each pair of groups yields a single result, rather than the unbounded cartesian product of the two, constraining the output but enabling privacy guarantees.

3.3. The partition operator

Theorem 4 tells us that structurally disjoint queries cost only the maximum privacy differential, and we would like to expose this functionality to the analyst. To that end, we introduce a `Partition` operation, like `GroupBy`, but in which the analyst must explicitly provide a set of candidate keys. The analyst is rewarded with a set of `PINQueryable` objects, one for each candidate key, containing the (possibly empty) subset of records that map to the each of the associated keys. It is important that PINQ does not reveal the set of keys present in the actual data, as this would violate differential privacy. For this reason, the analyst must specify the keys of interest, and PINQ must not correct them. Some subsets may be empty, and some records may not be reflected in any subset.

The `PINQAgent` objects of these new `PINQueryable` objects all reference the same source `PINQAgent`, of the source data, but following Theorem 4 will alert the agent only to changes in the maximum value of `epsilon`. The agents share a vector of their accumulated `epsilon` values since construction, and consult this vector with each update to see if the maximum has increased. If so, they forward the change in maximum. If the maximum has not increased, they accept the request.

The difference between the uses of `GroupBy` and `Partition` in PINQ can be seen in the following two queries:

Q1. How many ZIP codes contain at least 10 patients?
Q2. For each ZIP code, how many patients live there?

For Q1, a `GroupBy` by ZIP, a `Where` on the number of patients, and a `Count` gives an approximate answer to the exact number of ZIP codes with at least 10 patients. For Q2, a `Partition` by ZIP, followed by a `Count` on each part returns an approximate count for each ZIP code. As the measurements can be noisy, neither query necessarily provides a good estimate for the other. However, both are at times important questions, and PINQ is able to answer either accurately depending on how the question is posed.

The `Partition` operator can be followed not only by aggregation but by further differentially private computation on each of the parts. It enables a powerful recursive descent programming paradigm demonstrated in Section 4, and is very important in most nontrivial data analyses.

3.4. Security issues in implementation

Although the stability mathematics, composition properties, and definition of differential privacy provide mathematical guarantees, they do so only when PINQ's behavior is in line with our mathematical expectations. There are many important but subtle implementation details intended to protect against clever attackers who might use the implementation details of PINQ to learn information the mathematics would conceal. These are largely the result of user-defined code that may attempt to pass information out through side channels, either directly through disk or network channels, or indirectly by throwing exceptions or simply not terminating. PINQ's purify function gives the provider the opportunity to examine incoming methods and rewrite them, either by restricting the computations to those comprised of known-safe methods, or by rewriting the methods with appropriate guards. There are other issues and countermeasures in the full paper, and likely more unrecognized issues to be discovered and addressed.

4. APPLICATIONS AND EVALUATION

In this section, we present data analyses written with PINQ. Clearly not all analysis tasks can be implemented in PINQ (indeed, this is the point), but we aim to convince the reader that the set is sufficiently large as to be broadly useful.

Our main example application is a data visualization based on Web search logs containing IP addresses and query text. The application demonstrates many features of PINQ largely absent from other privacy-preserving data analysis platforms. These include direct access to unmodified data, user-supplied record-to-record transformations, operations such as `GroupBy` and `Join` on "sensitive" attributes, multiple independent data sets, and unfettered integration into higher-level programs.

For our experiments we use the DryadLINQ[15] provider. DryadLINQ is a research LINQ provider implemented on top of the Dryad[9] middleware for data parallel computation, and currently scales to at least thousands of compute nodes. Our test data sets are of limited size, roughly 100GB, and do not fully exercise the scalability of the DryadLINQ provider. We do not report on execution times, as PINQ's reasoning is an insignificant contribution, but rather the amount and nature of information we can extract from the data privately.

For clarity, we present examples written as if the data analyst is also the data provider, charged with assembling the source `PINQueryable` objects. In a real deployment, this assembly should be done on separate trusted infrastructure.

4.1. Data analysis: Stage 1 of 3

We start with a simple application of PINQ, approximating the number of distinct search users who have searched for an arbitrary query term. Our approach is just as it would have been in LINQ: we first transform the search records (comma-delimited strings) into tuples (string arrays) whose fields have known meaning, then restrict the data to records with the input search query, then group by the supplied IP address to get a proxy for distinct users, then count the remaining records (groups of string arrays). The program is reproduced in Example 4.

EXAMPLE 4. MEASURING QUERY FREQUENCIES IN PINQ.

```
// prepare data with privacy budget
var agent = new PINQAgentBudget(1.0);
var data  = new PINQueryable<string>(rawdata, agent);

// break out fields, filter by query, group by IP
var users = data.Select(line => line.Split(',')) 
                .Where(fields => fields[20] == args[0])
                .GroupBy(fields => fields[0]);

// output the count to the screen, or anywhere else
Console.WriteLine(args[0] + ":" + users.Count(0.1));
```

This relatively simple example demonstrates several important features of PINQ. The input data are text strings; we happen to know a priori that they are comma delimited, but this information plays no role in the privacy guarantees. The filtering is done against an analyst-supplied query term, and may be frequent or infrequent, sensitive or insensitive. To get the set of distinct users we group using the logged IP address, clearly highly sensitive information. Despite these uncertainties about the analysis, the differential privacy guarantees are immediately quantifiable.

4.2. Data analysis: Stage 2 of 3

Our program as written gives the count for a single query, and if the analyst wants additional counts they must run the program again. This incurs additional privacy cost, and will be unsuitable for extracting large numbers of query counts.

Instead, we can rewrite the previous program to use the `Partition` operator to permit an arbitrary number of counts at fixed privacy cost. Rather than filter records with `Where`, we use the same key selection function and an input set of query strings to `Partition` the records. Having done so, we iterate through each of the queries and associated parts, grouping the records in each by IP address. To further enrich the example, we then partition each of these data sets by the number of times each IP address has issued the query, before producing a noisy count (see Example 5).

EXAMPLE 5. MEASURING MANY QUERY FREQUENCIES IN PINQ.

```
// prepare data with privacy budget
var agent = new PINQAgentBudget(1.0);
var data  = new PINQueryable<string>(rawdata, agent);

// break out fields, but partition rather than filter
var parts = data.Select(line => line.Split(','))
                .Partition(args, fields =>
fields[20]);

foreach (var query in args)
{
  // use the searches for query, grouped by IP address
  var users = parts[query].GroupBy(fields => fields[0]);

  // further partition by the frequency of searches
  var freqs = users.Partition(new int[] { 1,2,3,4,5 },
                              group => group.Count());

  // output the counts to the screen, or anywhere else
  Console.WriteLine(query + ":");
  foreach (var count in new int[] { 1,2,3,4,5 })
    Console.WriteLine(freqs[count].Count(0.1));
}
```

Because we use `Partition` rather than multiple `Where` calls, the privacy cost associated with the program can be seen by PINQ to be only the maximum of the privacy costs of each of the loops, exactly the same cost as in Example 4.

Table 1 reports the measurements of a few query strings taken over our data set. Each reported measurement is the exact count plus Laplace noise with parameter 10, corresponding to standard deviation $10\sqrt{2}$. For most measurements this error is relatively insignificant. For some measurements it is significant, but nonetheless reveals that the original value is quite small.

4.3. Data analysis: Stage 3 of 3

We now expand out our example program from simple

Table 1. Numbers of Users Searching for Various Terms, Broken Out by Number of Times They Searched.

	Freq 1	Freq 2	Freq 3	Freq 4	Freq 5
GOOGLE	356,743	108,336	45,363	25,092	14,347
YAHOO	140,966	42,379	17,624	9671	5,707
BAIDU	300	79	29	26	9
AMAZON	16,798	3,376	808	378	132
EBAY	100,338	26,205	9,564	4,065	2,604
CNN	25,442	7,492	2,899	1,658	919
MSNBC	7,828	2,496	849	565	283

reporting (a not uncommon task) to a richer analysis application. Our goal is to visualize the distribution of locations of searches for various search queries. At a high level, we will transform the IP addresses into latitude–longitude pairs, by joining with a second proprietary data set, and then send the coordinates to a visualization algorithm borrowed from the work of McSherry and Talwar.[12] Although we will describe the visualization algorithm at a high level, it is fundamental that PINQ provides privacy guarantees even without the knowledge of what the algorithm plans to do with the data.

Starting from the prior examples, in which we have partitioned the data sets by query and grouped the results by IP address, we now demonstrate a fragment that will let us transform IP addresses into latitude–longitude coordinates. We use a second data set `iplatlon` whose entries are IP addresses and corresponding latitude–longitude coordinates. We join these two data sets, using the IP addresses in each as keys, resulting in a lat-lon coordinate pair in place of each group of searches. Example 6 contains the code for this `Join` transformation.

Example 6. Transforming IP addresses to coordinates.

```
// ... within the per-query loop, from before ...

// use the searches for query, group by IP address
var users = parts[query].GroupBy(fields => fields[0]);

// extract IP address from each group, and match
var coords = users.Join(iplatlon,
                        group => group.Key,
                        entry => entry.IP,
                        (group, entry) => entry.LatLon);
```

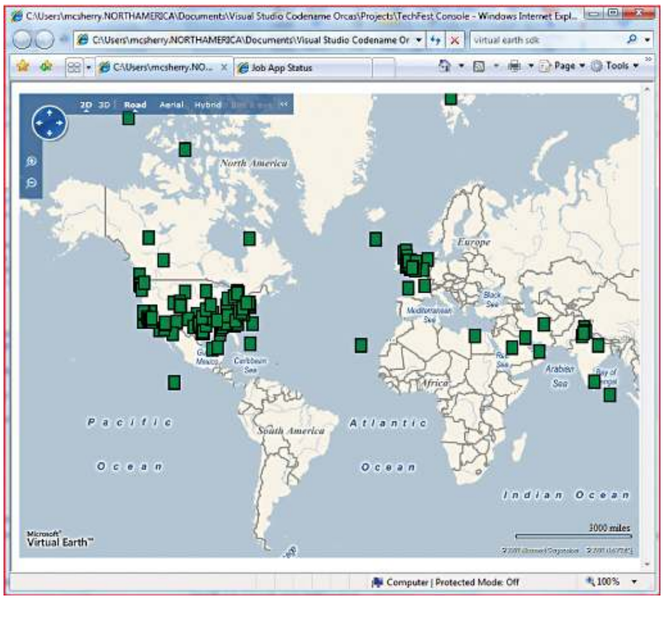
Recall that `Join` in PINQ only reports pairs that result from unique matches. In this program, we know that each IP occurs as a key at most once in `users`, as we have just performed a `GroupBy` with this field as the key. In the second data set, we assume (perhaps wrongly) that there is one entry per IP address. If this is not the case, or if we are not sure, we could also group the second data set by IP address and use the first lat-lon entry in each group. This is arguably more robust, but results in an additional factor of two in the privacy cost against the `iplatlon` data set; we would like to avoid this cost when we know we can safely do so.

Finally, our algorithm takes the list of lat-lon coordinates of the IPs searching for the input search query, and invokes a `Visualization` subroutine, whose implementation is not specified here. An example for the query "cricket" can be seen in Figure 4. Readers who are not entirely sure how or why this routine works are in roughly the same situation as most data providers. We have no intuition as to why the computation should be preserve privacy, nor is any forthcoming. Nonetheless, as the routine is only provided access to the data through a `PINQueryable`, we are assured of differential privacy guarantees even without understanding the algorithm's intent or implementation.

Support for such "modular design" of privacy algorithms is an important enabler for research and development, removing the need for end-to-end understanding of the computation. This is especially important for exploratory data analysis, where even the analysts themselves may not know the questions they will need answered until they start asking them. Removing the requirement of whole-program understanding also enables proprietary data analyses, in which an analyst may not want to divulge the analysis they intend to conduct. While the execution platform clearly must be instructed in the computations the analyst requires, the data provider does not need to be informed of their specifics to have privacy assurances.

Our second data set raises an interesting point about alternate applications of differential privacy. While the operation we perform, mapping IP addresses to latitude–longitude pairs, is essentially just a complicated `Select`, the data set describing the mapping is proprietary. Each record in the data set required some investment of effort to produce, from which the owners presumably hope to extract value; they may not want to release the records in the clear. Using this data set through PINQ prevents the dissemination of individual records, preserving the value of the data set while still permitting its use in analyses. Similarly, many organizations have data retention policies requiring the deletion of data after a certain amount of time. Ensuring that this deletion happens when analysts are allowed to create their own copies of the data is effectively impossible. Again, PINQ allows analysts to use such data without compromising the organization's obligations.

Figure 4. Example output, displaying a representative distribution of the latitude–longitude coordinates of users searching for "cricket." The computation has differential privacy not because of properties of the output itself, a quite complicated artifact, but because of the manner in which it was produced.



5. RELATED WORK

The analysis of sensitive data under the constraints of confidentiality has been the subject of a substantial amount of prior research; for an introductory survey we recommend the reader to Adam and Wortmann,[1] but stress that the field is still very much evolving. For an introduction to differential privacy we recommend the reader to Dwork.[6]

While PINQ is the first platform we are aware of providing differential privacy guarantees, several other interactive data analysis platforms have been proposed as an approach to providing privacy guarantees. Such platforms are generally built on the principle that aggregate values are less sensitive that individual records, but are very aware that allowing an analyst to define an arbitrary aggregation is very dangerous. Various and varying criteria are used to determine which aggregates an analyst should be able to conduct. To the best of our knowledge, none of these systems have provided quantifiable end-to-end privacy guarantees.

Recent interest in differential privacy for interactive systems appears to have started with Mirkovic,[13] who proposed using differential privacy as a criteria for admitting analyst-defined aggregations. The work defines an analysis language (targeted at network trace analysis) but does not go so far as to specify semantics that provide formal differential privacy guarantees. It seems possible that PINQ could support much of the proposed language without much additional work, with further trace-specific transformations and aggregations added as extensions to PINQ.

Airavat[14] is a recent analogue of PINQ for Map-Reduce computations. The authors invest much more effort in hardening the system, securing the computation through the use of a mandatory access control operating system and an instrumented java virtual machine, as well as PINQ-style differential privacy mathematics. At the same time, it seems that the resulting analysis language (one Map-Reduce stage) is less expressive than LINQ. It remains to be seen to what degree the system level guarantees of Airavat can be fruitfully hybridized with the language level restriction used in PINQ.

6. CONCLUSION

We have presented "Privacy Integrated Queries" (PINQ), a trustworthy platform for privacy-preserving data analysis. PINQ provides private access to arbitrarily sensitive data, without requiring privacy expertise of analysts or providers. The interface and behavior are very much like that of Language Intergrated Queries (LINQ), and the privacy guarantees are the unconditional guarantees of differential privacy.

PINQ presents an opportunity to establish a more formal and transparent basis for privacy technology and research. PINQ's contribution is not only that one can write private programs, but that one can write only private programs. Algorithms built out of trusted components inherit privacy properties structurally, and do not require expert analysis and understanding to safely deploy. This expands the set of capable users of sensitive data, increases the portability of privacy-preserving algorithms across data sets and domains, and broadens the scope of the analysis of sensitive data.

6.1. Availability

The prototype of PINQ used for the experiments in this paper, as well as further example programs and a brief tutorial, are available at http://research.microsoft.com/PINQ.

Acknowledgments

The author gratefully acknowledges the contributions of several collaborators. Ilya Mironov, Kobbi Nissim, and Adam Smith have each expressed substantial interest in and support for privacy tools and technology usable by nonexperts. Yuan Yu, Dennis Fetterly, Úlfar Erlingsson, and Mihai Budiu helped tremendously in educating the author about LINQ, and have informed the design and implementation of PINQ. Many readers and reviewers have provided comments that have substantially improved the presentation of this paper.

References

1. Adam, N.R., Wortmann, J.C. Security-control methods for statistical databases: A comparative study, ACM Comput. Surv., 21, 4 (1989), 515–556.
2. Barak, B., Chaudhuri, K., Dwork, C., Kale, S., McSherry, F., Talwar, K. Privacy, accuracy, and consistency too: a holistic solution to contingency table release, in PODS (2007), 273–282.
3. Barbaro, M., Zeller Jr., T. A face is exposed for AOL searcher no. 4417749, The *New York Times*, August 9, 2006.
4. Blum, A., Dwork, C., McSherry, F., Nissim, K. Practical privacy: The SuLQ framework, in PODS (2005), 128–138.
5. Dwork, C. Differential privacy, in ICALP (2006), 1–12.
6. Dwork, C. A firm foundation for private data analysis, Communications of the ACM, Association for Computing Machinery, Inc., 2010.
7. Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., Naor, M., Our data, ourselves: Privacy via distributed noise generation, in EUROCRYPT (2006), 486–503.
8. Dwork, C., McSherry, F., Nissim, K., Smith, A. Calibrating noise to sensitivity in private data analysis, in TCC (2006), 265–284.
9. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks, in EuroSys. ACM (2007), 59–72.
10. McSherry, F. Privacy integrated queries: an extensible platform for privacy-preserving data analysis, in SIGMOD Conference (2009), 19–30.
11. McSherry, F., Talwar, K. Mechanism design via differential privacy, in FOCS (2007), 94–103.
12. McSherry, F., Talwar, K. Synthetic data via differential privacy, Manuscript.
13. Mirkovic, J. Privacy-safe nework trace sharing via secure queries, in NDA (2008).
14. Roy, I., Setty, S.T., Kilzer, A., Shmatikov, V., Witchel, E. Airavat: Security and privacy for mapreduce, in NSDI Conference (2010).
15. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U, Gunda, P.K., Currey, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language, in OSDI (2008).

Frank McSherry (mcsherry@microsoft.com), Microsoft Research, SVC, Mountain View, CA.