**SmartNoise SQL API and SmartNoise Core**

The SmartNoise SQL API allows analysts to query large databases for differentially private results. The API rewrites incoming SQL queries, issues aggregate queries against the database engine, then applies differential privacy to the aggregated results, using SmartNoise Core Library.

This document highlights some open design questions for further integration with SmartNoise Core Library.

**SQL API Characteristics**

All SELECT statements must include aggregation expressions, such as SUM, AVG, COUNT, MIN. Each SELECT statement may contain multiple *output columns*, each of which represents an aggregate statistic over the data. Output columns may refer to different input columns, or may process the same input columns in different ways. For example:

SELECT AVG(income), MAX(income), COUNT(User)

Has two different output statistics computed from the same source column, and a third output column selected from a different source column. Each output column incurs a privacy cost.

A SQL GROUP BY statement always partitions data disjointly. Each output statistic is computed once per each grouping key, hereafter referred to as a *partition*. For example:

SELECT AVG(income) GROUP BY married

Will result in two average incomes being reported; one for married and one for unmarried people. Because the GROUP BY query disjointly partitions data, we pay the same privacy cost for a query, regardless of the number of partitions.

**Privacy Cost**

The privacy cost depends on the number of columns in the query. For example, if the per-column epsilon is set to 0.1, and the query asks for:

SELECT COUNT(UserID), COUNT(DISTINCT ProductID)

The total cost would be 0.1 x 2, or 0.2 epsilon. Some expressions use a multiple of the per-column epsilon. For example:

SELECT AVG(income)

Would use 0.2 epsilon, because it is converted under the covers to:

SELECT SUM(income) / COUNT(income)

The parser re-uses noisy answers, however, so the following query would use only 0.2 epsilon, despite having two more columns than the query above:

SELECT AVG(income), COUNT(income), COUNT(income) \* 100

**Censoring Partitions**

The API will censor partitions that are too small to protect privacy. There are several supported censoring algorithms, and they all consume privacy budget. The simplest censoring routines compute the distinct count of users per-partition under the covers, and threshold using the noisy count.

Because the noisy distinct count is computed under the covers in this case, these counts may be available to the analyst “for free”. For example, these two queries have the same cost:

SELECT educ GROUP BY educ;

SELECT educ, COUNT(DISTINCT users) GROUP BY educ;

**User Contribution**

In some rowsets, users may appear more than once. Aggregate queries need to scale the sensitivity of noise to account for the number of times a user may appear.

Although the GROUP BY semantics ensure that *rows* are partitioned disjointly, it is not always the case that users will be partitioned disjointly.

We can categorize queries based on two parameters that describe the user contribution:

* *tau* – The maximum number of times a user can appear in each partition
* *Delta* – The maximum number of partitions a user can appear in

Here are some examples of queries, characterized by user contributions:

**tau = 1, Delta = 1**

SELECT AVG(income) FROM PUMS GROUP BY married

This type of query is typical for row-privacy, common in social science and other tidy datasets.

**tau = \*, Delta = 1**

SELECT Region, SUM(TotalPrice) FROM CustomerOrder GROUP BY Region

In this query, each user is assigned to only one region, so each user can appear in only one partition. However, each user may have multiple orders.

**tau = 1, Delta = \***

SELECT ProductCategory, COUNT(DISTINCT CustomerID) FROM Order

In this query, each user may have purchased products from multiple categories, but each user can impact the count of each bin by at most 1.

**tau = \*, Delta = \***

SELECT url, COUNT(DISTINCT User), SUM(DwellTime)

This query can be used to rank web sites by unique users and engagement time. Each user may browse several web sites, and the SUM of DwellTime may include many visits from the same user to a single web site.

**Inferring and Enforcing User Contribution**

The SQL API has limited ability to infer that a query conforms to specific *tau* and *Delta* bounds, by validating the form of the query and using information from metadata. In many cases, the validator will not be able to infer these bounds, or may need to assume bounds that are too large.

The SQL API is capable of reservoir sampling to enforce *tau*, and can also sample to enforce the maximum number of times a user can appear across all partitions, but it is not straightforward to enforce a bound on *Delta*. In practice, *Delta* is often either 1 (due to query semantics) or is unbounded, being as large as the number of partitions.

**Implementation**

The API currently processes each partition sequentially, in streaming fashion, because the number of requested partitions may be very large.

For each partition, all output columns are computed at the same time, in memory. The rows for each partition are not available to the API. The API must work with aggregate statistics returned by the database engine. The Gaussian mechanism is used to add noise to all source data. The Gaussian mechanism is used for simplicity in composition across repeated queries.

**Open Questions**

1. Can we swap out some of our routines with more accurate routines from Core?
   1. MEAN/AVG
   2. COUNT (using different mechanisms)
   3. Expressions (e.g. SUM(income) \* 100)
2. If we use a mix of mechanisms (e.g. one mechanism for counts, another for sums), is there a way to get good composition across repeated queries?
3. Would it make sense to use different mechanisms for different partition sizes if metadata hints are available?
4. Can we add support for additional statistics such as MAX, MIN, MEDIAN, using only quantities that can be returned in a GROUP BY (access to row-level data is not possible).
5. How can we obtain accuracy bounds for columns? Are there cases (such as MEAN) where accuracy will vary depending on the partition size?
6. Calling mechanisms in Core is very fast, due to FFI. However, we presumably need to use the graph if we want to handle more complex expressions and built-ins like MEAN or MEDIAN.
   1. Can the graph be called with caller-supplied sensitivity (for both the user contribution and column values) and aggregates for these statistics?
   2. Is there significant calling overhead for the graph versus mechanisms?