Limited SQL Semantics for Differential Privacy

This document describes a subset of SQL semantics that can be used to safely build differentially private reports which target a variety of backends, including non-SQL systems. In addition, the document describes the basic steps that are needed to map from parsed SQL language syntax to an intermediate format, in order to support queries written in SQL syntax.

Input

- **Records**: (required) A list of tuples which may contain private data. We refer to each tuple as a 'record', and we refer to each positional entry of the tuple as a 'column'. Each record is a tuple of values, which may include values of type int, float, boolean, string, or datetime. Individual tuples in a list must match in types, but entries in the same tuple do not need to match. For example, a valid list of records might include tuples of type ⟨float, float, string, bool⟩, but a list of records with a mix of ⟨float, string, bool, string⟩ and ⟨string, bool, float, string⟩ would be invalid. One of the columns must represent the identity of the private item to be protected (e.g. user_id). Columns in tuples may be referenced by name.
- **Metadata**: (required) The metadata specifies which column represents the private key, and specifies type information about the source tuples, such as min and max for numeric columns and cardinality for columns to be used in grouping expressions.
- **Predicates** (optional): A list of predicates used to filter a subset of records. Predicates are simple Boolean expressions, but may include transforms of the source tuples. For example, a simple predicate might filter records where $t[0] > 10$, meaning tuples where the first entry is greater than 10. A more complex predicate might transform a tuple value, $(t[0] \,\hat{}\, 2)/2 > 10$, or transform multiple values, $(t[0] - t[1]) > 0$.
- **Inner Transforms** (optional): A list of expressions used to transform the tuples before aggregation. Transforms always result in the same number of records, but tuples may change in type.
- **Aggregate Expressions** (optional): A list of grouping expressions used to partition the source records before summarizing. The records will be grouped by the cross-product of all distinct aggregate expression values. If aggregate expressions are omitted, the entire list of tuples will be aggregated to a single partition.

- **Summary Functions** (required): A list of summarization functions to apply over each aggregated partition. Examples include Count, Sum, Avg, Var, and so on. Tuples from this step may include only summary function results or grouping keys from aggregate expressions. This step will have number of tuples equal to the cross-product of distinct aggregates.
- **Outer Transforms** (optional): A list of transforms to apply to summary tuples.
- **Outer Predicates** (optional): A list of predicates to be used to filter the aggregated results.
- **Order Expressions** (optional): Specifies how to sort the final output.
- **Truncate Expression** (optional): Specifies how many rows of sorted output to return.

There are 3 required and 5 optional inputs. These inputs will be referenced throughout the document.

Input Restrictions

Summary functions are restricted to functions for which we know how to compute private answers.

We support only one aggregation step. Summary functions are required, and may be surrounded by only one stage of non-summary transforms before and after summarization. For example, the following are permitted:

- 10 + Sum(t[0]) # outer transform
- Sum(t[0] ^ 2)   # inner transform
- 10 + Sum(t[0] / 7)
- 10 + Sum((t[0]/7) + (t[1]/3))

The following are prohibited, despite being possible in SQL:

- Count(10 + Sum(t[0]))
- Sum(Count(t[0] / 7))

Predicates only apply to source tuples, and cannot be used to filter based on the results of aggregation, inner transform, or outer transform.

The source records may be constructed by a series of joins, aggregates, and other operations independently of the differentially private processing. The differentially private query must receive a single set of tuples, regardless of how they are created. To ensure differential privacy, the source records must be created in a way that ensures no private key contributes to more than one partition in the source records, and the private key must be included as a column in the source records.

Exact Query Semantics

The query processor computes exact (non-private) results in the following order:

1. Filter: The records are first filtered to return only records that match the desired predicates. The number of records may change, but the number columns and column types and values of each record do not change.
2. Projection: Each record is transformed according to any per-record transformations. The number of records does not change, but the number of columns and values may change.
3. Aggregation: Groups of records are partitioned according to the specified partition keys.
4. Summarization: The specified summary functions are computed over the records in each partition. The number of records will now equal the number of partitions, and the columns may change.
5. Projection: Each summarized record is evaluated against any remaining outer (non-aggregate) expressions. The number of records does not change, but the columns may change.
6. Filter: The final aggregated results are filtered according to the outer predicates.
7. Sort: The columns are sorted according to the values in the final projection step.
8. Truncate: The results are truncated to return the number of aggregated rows requested.

Private Query Processing

1. Clamp: The clamp operation clamps all source tuples to conform to sensitivity bounds specified in the metadata. This results in the same number of tuples, with the same types. Alternately, the clamp operation can be replaced with a filter operation that drops outliers. Result includes private key.

2. Filter: Drop all clamped records that do not match the predicates. Results in record set with same shape but may have fewer records. Result includes private key.
3. Sample: Perform reservoir sampling to bound the contribution of each private key value. Result will be record set with same shape but may have fewer records. Result includes private key.
4. Projection: Compute all inner expressions to produce a new set of tuples with the same number of records, but may be different types and number of columns.
5. Aggregation: Partition the records based on aggregation expressions.
6. Summarization: Compute all summary functions on records within each partition. Additionally, count the number of distinct private keys contributing to each partition. This will result in number of tuples equal to the number of partitions, and may change the types and number of columns.
7. Noise: Use privacy mechanisms to add noise to summary results. Shape of record set will remain the same.
8. Filter: Drop all partitions where count of distinct keys is below privacy threshold.
9. Projection: Compute any outer expressions that operate on summarized values.
10. Filter: Drop output aggregates based on outer predicates.
11. Sort: Sort operates over the noise results.
12. Truncate: Return the number of sorted rows specified.

Simplified Example

First consider a simple example, where only the minimum required inputs are provided:

- **Records**: a list of tuples with one column ⟨key⟩
- **Metadata**: no numeric value, so metadata consists only of key column specification
- **Summary Functions**: Count(key)

This is equivalent to a SQL query like: `SELECT COUNT(key) FROM relation;`

Using the numbering for "Private Query Processing" above, we need to perform only 4 steps:

3. Sample
6. Summarize

7. Noise
8. Filter

We now elaborate on each of these functions:

The sample function is a generic *filter* function, and only needs to know which column is the key, and how many times each key may appear.

```
Sample {
    Inputs:
        records: List[⟨key, ...⟩] # the records
        tau: Int          # integer bounds on private key
    return reservoir_sample(records, count(key) <= tau)
}
```

The summary functions are generic *reduce* functions, and simply take a vector of some values and return a result. Sum, Count, Min, Max, etc. are available in every programming language and data processing platform.

For purposes of translating SQL semantics, some summary functions can specify whether null values are included, using quantifiers 'all' versus 'distinct'.

The noise function is a generic *map* function, which only needs to know how much noise to add to each mapped output. The noise is scaled by the sensitivity of the column. In the case of counts, the sensitivity is 1 scaled by the maximum contribution of each key, which is our parameter *tau*.

```
Noise {
    Inputs:
        summaries: List[val1, ...]
        sensitivities: List[scale1, ...]
    return List[val + Random(0, sens) for val, sens in zip(summaries, sensitivities)]
}
```

In our simple case, the summaries parameter is a list with one row and one column, consisting of the exact count (computed by the summary step). Our sensitivities list also has one row and one column, with a single value of *tau*. We add random noise scaled to our sensitivity.

The last step is another *filter* function, which simply drops any rows with too few keys. This function needs only one parameter, representing the threshold, for example *tau * tau*.

To recap, our processing involves a filter, a summarize, a map, and then a filter. The summarize function needs no privacy-specific parameters, and the other functions need only minimal information. The computation can be thought of as:

result = *relation*.filter(max=*tau*).summarize().map(scale=*tau*).filter(thresh=*tau***tau*)
The final filter operates on a single row, since there is only one row of output. Therefore, the final filter step will result in the answer either being shown or not shown. In examples with aggregation we will look at later, the filter step may show some partitions while dropping other.

Because these 4 functions are fairly simple and generic, we do not need to consider them further in the remainder of this document.

Example with Clamping

We now consider an example where we summarize over a value with a sensitivity greater than one.

- **Records**: a list of tuples with two columns ⟨key, value⟩
- **Metadata**: specifies the key column, and also the min and max of the value column, for example [5,30]
- **Summary Functions**: Sum(value)

This is equivalent to a SQL query like: `SELECT SUM(value) FROM relation;`

1. Clamp
3. Sample
6. Summarize
7. Noise
8. Filter

In this example, we have one new function to define. The clamp function is a generic *map* function.

```
Clamp {
    Inputs:
        vals: List[val1, ...]
        min: Int|Float
        max: Int|Float
    return vals.map(v => min if v < min, max if v > max, else v)
}
```

There is one min and one max that are applied to all values in the column. In some applications, we might prefer to implement as a filter, dropping any records that are outside the min and max bounds.

Note that the sensitivity parameter passed to the noise function in this example with be the range of the value (in this example, **|30 - 5|**, or 25) multiplied by *tau*.

Also note that we need to keep track of Count(key) internally, to allow filter step 9 to operate.

Aggregates

This is the first example we consider that includes an optional input. For this example, we include an aggregation expression:

- **Records**: a list of tuples with three columns ⟨key, value, group_key⟩
- **Metadata**: specifies the key column, and also the min and max of the value column, for example [5,30]
- **Aggregate Expression**: ⟨group_key⟩
- **Summary Functions**: Sum(value)

This is equivalent to SQL:
`SELECT SUM(value) FROM relation GROUP BY group_key;`

1. Clamp
3. Sample
5. Aggregation

6. Summarize
7. Noise
8. Filter

The aggregation function is a generic reduce function, and does not need any information about differential privacy. Aggregation reducers are available in all programming languages and data processing platforms.

```
Aggregate {
    Inputs:
        records : List[⟨key, ..., group_key⟩
        expr: <group_key, ...>
    return List[⟨group_key1, List[⟨key, ...⟩, ⟨key, ...⟩, ...]⟩,
                ⟨group_key2, List[⟨key, ...⟩, ⟨key, ...⟩, ...]⟩,
                 ...]
}
```

Instead of a single row, there will now be one row of output per distinct value of *group_key*. All of the steps after aggregation (in this case, summarize, noise, and filter) will now be performed on each partition of records. We now maintain internal Count(*key*) per-partition, so the final filter step can drop partitions with insufficient keys.

Predicates and Order

We next consider two more optional inputs.

- **Records**: a list of tuples with three columns ⟨key, value, group_key⟩
- **Metadata**: specifies the key column, and also the min and max of the value column, for example [5,30]
- **Predicates**: (value > 10)
- **Aggregate Expression**: ⟨group_key⟩
- **Summary Functions**: Sum(value)
- **Order Expressions**: ⟨group_key descending⟩

In SQL, this would be:

```
SELECT Sum(value) FROM relation WHERE value > 10 GROUP BY group_key ORDER
BY group_key DESC;
```

To support these, we need to define two new functions.

1. Clamp
2. Filter
3. Sample
5. Aggregation
6. Summarize
7. Noise
8. Filter
10. Sort

The predicate selection step is a generic *filter* function, and the sort step is a generic sort function. The predicates must be applied after clamping and before aggregation, while the sort must be applied after all other steps. There is nothing specific to privacy in either step, and implementations of both filter and sort are widely available, so we will not consider these further.

Outer Transforms

For the remainder of this document, we will skip enumeration of the inputs defined at the beginning of the document and use SQL syntax. Consider an example like this:

```
SELECT Sum(value) + 10 FROM relation;
```

1. Clamp
3. Sample
6. Summarize
7. Noise
8. Filter
9. Projection

This example is an "outer transform", because the arithmetic expression applies to the final sum. The outer transform is a *map* function, mapping noisy values to outputs. Recall from earlier than the noise function requires sensitivity information to properly protect privacy. Because differential privacy is immune to pos-processing, we can apply arbitrary expressions to the noisy summaries without affecting privacy.

Inner Transforms

Now consider a SQL query like this:

```
SELECT SUM(value * value) FROM relation;
```

1. Clamp
3. Sample
4. Projection
6. Summarize
7. Noise
8. Filter

This projection represents a *map* function as well, but the values are mapped before aggregation and summarization. Because these transformations occur before the noise function, they may affect the sensitivity of the inputs to noise. Because of this, we can only support inner transforms for which we know how to compute appropriate sensitivity.

Translating Functions: Outer and Inner Transforms

In some cases, we will prefer to construct noisy summaries from outer expressions. For example, if we want to compute:

```
SELECT AVG(value) FROM relation;
```

We will not compute the exact average and then add noise scaled to the range of value. Instead, we might compute the exact sum and the exact count, and divide the two:

```
SELECT SUM(value) / COUNT(value) FROM relation;
```

As we discussed regarding outer transforms, the expression above is applied after noise is added, and immunity to post-processing ensures that we maintain privacy.

Some functions may likewise be possible to translate to inner expressions, which are applied before noise is added. In these cases, we must ensure that sensitivity is calculated properly in order to scale noise at the aggregation step.

For example, consider a variance calculation:

```
SELECT VAR(value) FROM relation;
```

Rather than computing the exact variance and then adding noise scaled to the range of variance, we want something that preserves privacy with better accuracy. The variance is defined as the expected value of the squared difference between each value and the mean. This can be translated to a combination of inner and outer expressions:

```
SELECT AVG(value * value) - (AVG(value) * AVG(value)) FROM relation;
```

with the Avg() computations in turn being translated into outer expressions.

Implementation Overview

To review, we have specified the 10 functions necessary to implement a limited subset of SQL semantics with differential privacy:

```
    clamped = relation.map(v => bounded(v))
    filtered = clamped.filter(where predicates)
    sampled = filtered.filter(reservoir sample)
    projected = sampled.map(v => inner_expr(v))
    aggregated = projected.reduce(group_by)
    summarized = aggregated.summarize(by key)
    noisy = summarized.map(v => noisy(v))
    private = noisy.filter(key_count > thresh)
    processed = private.map(v => outer_expr(v))
    final = process.sort()
```

These are all very simple, and can be implemented reliably in a variety of languages. In some cases, it may be desirable to split execution across implementations. For example, modern database engines are very good at computing exact aggregates, so one approach would be to run everything up to ‘aggregated’ as SQL, and then post-process ‘noisy’ through ‘final’ in the data access layer. In other cases, the entire process can be implemented in the database engine, with user-defined functions to add noise.

Alternately, the entire process can be implemented outside the database engine; for example using map/reduce/filter style processing over CSVs in JavaScript, Python, or Scala; or perhaps using custom C++ implementations of all 10 operations running in a secure SGX enclave.

Because the operations are fixed, and require only a limited number of parameters, it is possible to encode operations as a simple tree structure with required parameters attached, and easily emit execution plans for various backends, such as PostgreSQL or SQL Server, or Spark.

SQL Parser

Completely independent of the intermediate representation and supported execution backends, we want to support construction of queries using familiar SQL language statements. This requires a parser that can safely parse SQL, validate that statements adhere to the supported subset and source relation requirements, and perform any pre-processing necessary to build the intermediate format; for example, converting AVG and VAR to the equivalent outer and inner expressions.

Discussion of the SQL syntax parser is beyond the scope of this document. However, the input system must handle the following basic steps:

1. Validate that source relation (everything in FROM clause) adheres to requirements: includes private key at all subquery and join steps, and private key never included in more than one partition.
2. Validate that query includes only one set of aggregate expressions.
3. Convert known summary functions to preferred alternates (e.g. AVG and VAR)
4. Validate that SELECT clause includes only summary functions or GROUP BY keys.
5. Validate that all summary functions can safely compute sensitivity and bounds are known.
6. Transform to intermediate representation

When converting back from intermediate representation to SQL, the system must:

1. Add key_count to support final filter step
2. Generate platform-specific sampling step
3. Generate appropriate join semantics, e.g. USING or ON

4. Use platform-specific handling for booleans, literals, escaped identifiers, etc.

Rewriting Queries

We now describe the transformation steps in more detail.

1. First, rewrite the original query to become the outer query, which will be evaluated during post-processing.
    a. Convert all expressions to use SUM and COUNT.
    b. Push all group-by columns into select scope for child query. These may or may not be used in the select statement. These are needed for the child group by clause to work.
    c. Push all column expressions anywhere in select statement which are not in summary functions into child select scope. These column expressions may be children of other expressions, such as arithmetic expressions or math functions. Only the column expressions should be pushed to child scope, because all outer expression evaluation happens at this layer. Column expressions inside summary functions are not pushed separately here, because the entire summary function will be pushed through.
    d. Convert all SUM and COUNT to SUM, and push inner expression to read from SUM or COUNT that was pushed to child scope. For example, COUNT(Foo) becomes SUM(count_Foo), with the COUNT(Foo) pushed to child scope. We use SUM in this stage, because the group by clause is pushed to child scope. Throw error if any summary function has summary function descendants. For example, if the original query has something like COUNT(3 / SUM(Foo)), we cannot support this.
    e. Keep order by. Push group by to child scope.
2. This is the aggregation subquery. All outer expressions have been dropped, and will be evaluated in the parent. This query contains only aggregation columns and summary functions, which may include complex inner expressions.
    a. Select all columns which have been pushed through from parent scope, and add a column for SELECT COUNT(key) AS key_count, to allow thresholding during postprocessing.
    b. The group by clause must not contain the key.
    c. Convert COUNT and SUM to SUM again, and push through like before. We use sums again, because counts are aggregated at a per-key level prior to reservoir sampling, and we require keys to contribute to only one partition, ensuring that sum of counts is the count.

3. Aggregate at same group by level, but include key.
    a. Reservoir sample
    b. Push through all Column expressions from inner expressions. Only column expressions are pushed through, because inner expressions are evaluated at this layer.
    c. Push through aggregation columns and key to child scope so they are available for aggregation in this layer.
4. Clamp all numeric columns directly from the table. Do not clamp key column.