Differential privacy computations in data pipelines reference doc

The Google Anonymization team
Last update: Apr 14, 2021

Motivation

The goals of this document are

- explain and summarize high-level computations, which are already implemented in Privacy on Beam (Go) and as an extension to ZetaSQL (C++).
- be a reference for implementation for those computations.
- help onboarding new people who would work on relevant projects.

This document elaborates on the technical approach outlined in the paper Differentially Private SQL with Bounded User Contribution. It does so in a language independent way, including references to DP theory, an explanation of computation blocks, computation graphs for pipelines, and discussion of some pipeline nuances. It suggests some generalizations.

This doc will be made public, it will be used as a reference doc for a collaboration project between OpenMined and Google.

Assumed knowledge for reading this document

1. Basic knowledge of differential privacy (DP). A good introduction is these blog posts.
2. Basic knowledge of a pipeline framework (like Apache Beam or Apache Spark).
3. Basic knowledge of SQL (it's used for explaining some computations).
4. Basic knowledge of linear algebra (vectors, vector norms).

Structure of this document

First we set up the problem space, namely which differential private computations are covered in this document and their relationship to non-DP computations. We then introduce some necessary terminology, and discuss the general structure of the DP computations.

Second, we describe the building blocks of DP computations in detail - including different options for each building block, in which order these building blocks are applied, etc.

Third, we show computation graphs of some example DP computations.

Problem setup and general algorithm

The doc explains how to compute different metrics with Differential Privacy. Computing with DP inevitably changes the semantics of queries. Let's start from non-DP queries.

Non-DP queries

We'd like to compute aggregated metrics from a collection of objects of the same type in a pipeline framework (like Apache Beam). More specifically, we make the following assumptions.

1. The input **collection** might be of any size, possibly stored on many machines. For example, a table in SQL, a PCollection in Beam, or even an array stored in a single machine. There is no order among elements and it's not allowed to use indices for referencing elements.
2. **Elements** of the collection have the same type. They might be rows in SQL or objects in OOP. Each element corresponds to data from one individual, but a single individual can contribute multiple elements.
3. The collections support generic operations:
    a. **Map** applies some specified function to each element and returns a collection containing the results.
    b. **Combine** takes a collection of (key, value) elements. For each key, it applies a given function to all the values with that key, and returns the output of this function for each key. Typically, the function is an aggregation:, like count, sum, etc. For example, applying the sum combiner to the collection [(key1, 1), (key2, 3), (key1, 17)] would return [(key1, 18), (key2, 3)].
    c. **Join** takes collections **col1** (key, value1) and **col2** (key, value2) and returns a collection (key, (values1, values2)) for all keys which are present in **col1** or **col2**.
    d. **Union** takes 2 collections with elements of the same type and returns one collection which has all elements of both input collections.

In SQL terminology, we're interested in computations of the form:

```
SELECT aggr1(c1), aggr2(c2),... aggrn(cn)
FROM collection
[GROUP BY partition_key]
```

where **c1**, … **cn** are columns (or fields) of objects in the collection, **aggr1**,... **aggrn** are aggregation functions, and **partition_key** might be compound from multiple columns or even a value of some function.

Remark: Group by **partition_key** is optional, i.e. aggregations can also be computed over the whole dataset.

In this doc, we consider the following aggregation functions:

1. COUNT
2. COUNT_DISTINCT
3. SUM
4. MEAN
5. VARIANCE
6. PERCENTILE
7. MIN/MAX

For simplification purposes, we assume in the following discussion that exactly one aggregation function is computed and aggregation is by partition_key, i.e.

```
SELECT aggr(value)
FROM collection
GROUP BY partition_key
```

Running example

Let's consider a running example of this document: restaurant ratings. The records of this dataset are restaurant visits, and consist of **restaurant_id, user_id, visit date, rating**.

Some possible queries to this dataset are

1. Number of visits to all restaurants (global aggregation).
2. Number of visits per restaurant (group by restaurant_id).
3. Average rating per restaurant.
4. Number of visits per day (or month).

DP metrics

This doc will follow the method of computing DP metrics from Section 2 of the paper Differentially Private SQL with Bounded User Contribution, adapting the discussion to pipeline frameworks and digging into some details and nuances.

At a high-level, DP is about hiding the effect of contributions from single users. A single user can:

1. Contribute multiple records to a given partition.
2. Contribute to multiple partitions.
3. Influence whether a partition appears in the output.

As outlined in the paper, to address points 1 and 2, we need to apply contribution bounding, and to address point 3, we need to apply partition selection. Note that these are necessary in order to obtain DP guarantees, but they alone are not sufficient: the aggregations themselves also have to be modified to implement DP, usually through adding random noise. We come back to these considerations in the section on DP aggregation computations.

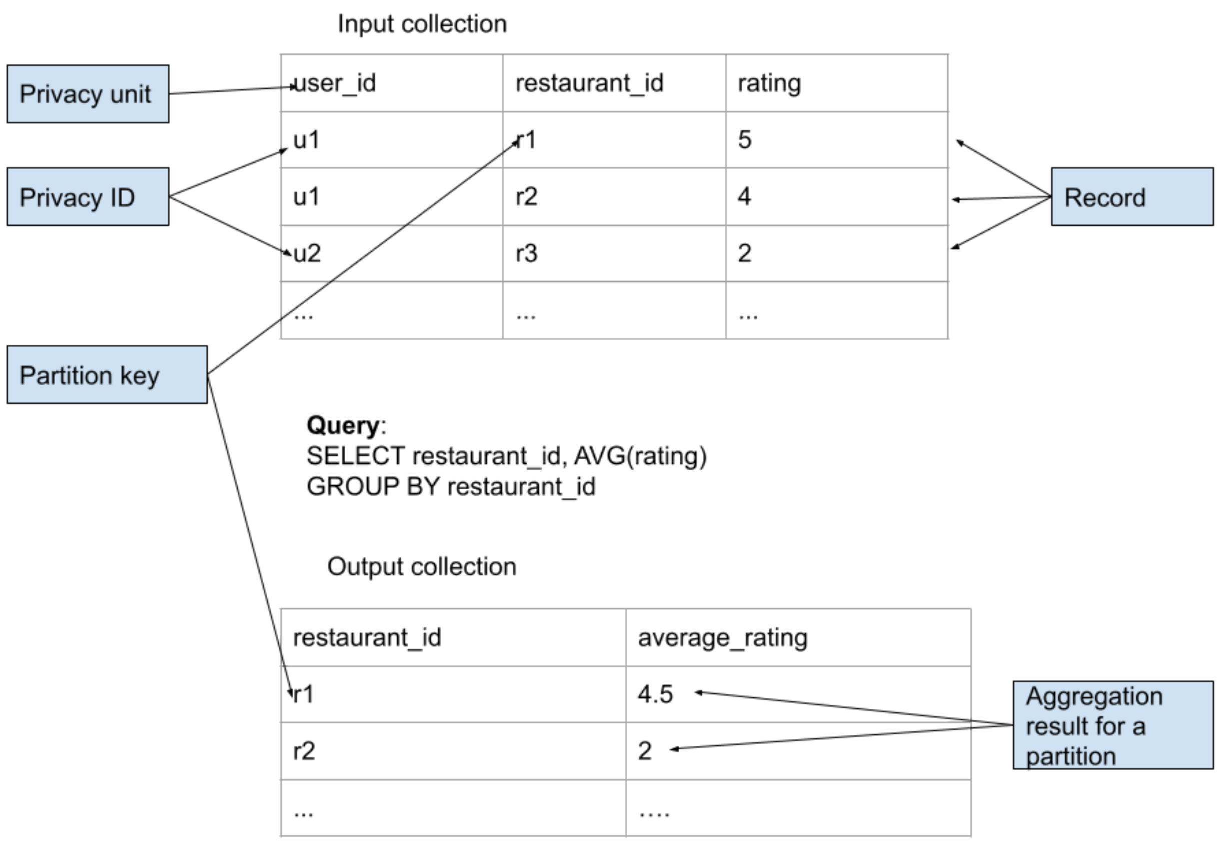
Note that to use DP, we need to understand the contribution of a single user, so we need privacy units (e.g. user ids) to be a part of the dataset.

Remark: Some datasets do not have any privacy units (if, for example, the data were collected in a de-identified way). It may still be possible to do proper DP computations in such a case if the data collection mechanism ensures that contributions are bounded (e.g. each individual can contribute only once). Such datasets can be processed through a tool like the one we describe by adding random unique identifiers and using them as privacy IDs.

This document focuses on “Per-partition aggregation, w/ privacy ids” as other cases (no privacy ids, global aggregation) can be easily mapped to this model.

Terminology

The following diagram illustrates some of the terminology used:



Input collection

user_id	restaurant_id	rating
u1	r1	5
u1	r2	4
u2	r3	2
...	...	...

Query:
SELECT restaurant_id, AVG(rating)
GROUP BY restaurant_id

Output collection

restaurant_id	average_rating
r1	4.5
r2	2
...	....

- A **privacy unit** is an entity that we're protecting with differential privacy. Often, this is a single individual. In that case, we would be guaranteeing that anyone observing the output of our aggregations could not use them to infer much about whether any of an individual's data was included. Different privacy units can protect different things. Examples:
    - individual_id - protects all actions of each individual
    - (individual_id, date) - protects actions of each individual per day. This privacy unit protects "less" than individual_id, because it protects only part of an individual's data.
    - (individual_id, restaurant_id) - protects visits of each individual to each restaurant.
- A **privacy ID** is an instance of a privacy unit. For example, if the privacy unit is (individual_id, date), then (individual ID = 1234, date = 2021-01-01) is a possible privacy ID.
- A **record** is a single object in an input collection.
- **A partition** is a subset of the data for which we return an aggregated result. Partitions are mutually exclusive; each record (though not each privacy ID) will be grouped into a single partition. Using the running example, if we calculate the average rating for each restaurant, restaurants are partitions.
- **Partition key** - the values that identify partitions. Examples of partition keys and corresponding partitions:
    - partition key = (date), each partition will contain all the activity on a certain day
    - partition key = (date, restaurant_id), each partition will contain all the visits to a particular restaurant on a particular day
- **Contribution bounding** is the process of limiting the contributions of a single privacy unit so that no privacy unit data has too much influence on the output aggregation result. There are 2 types of contribution bounding:
    - **Cross-partition contribution bounding** is a procedure to ensure that each individual contributes to a limited number of partitions.
    - **Per-partition contribution bounding** is a procedure to ensure that each individual's contribution to any single partition is bounded.
- **Partition selection (aka thresholding)** is a procedure which ensures that the list of partitions present in the output does not, in itself, break DP guarantees.
- **Privacy (loss) budget:** a pipeline may include multiple DP calculations, each of which has its own epsilon and delta. Due to the composition property of DP, we can treat the entire pipeline as a single DP operation with a certain epsilon and delta. The total epsilon and delta for the pipeline are its budget, and DP sub-computations within the pipeline consume parts of that budget.

Algorithm of DP metrics calculation in general

Computation of DP metrics typically involves the following steps:

1. Bound contributions
2. Perform partition selection
3. Compute aggregated metrics
4. Add noise to aggregated metrics

Remark: the order of the first three steps is not fixed. More discussions on trade-offs later.

Algorithm blocks in more details

Remark: this section covers the basic blocks. The list presented in this section isn't exhaustive.

Noise addition mechanisms

This section discusses the Laplace and Gaussian mechanisms. There is only the necessary minimum of information required for further discussion.

Let **q** be a query on datasets with individual data, which returns a **n** dimension vector for each dataset.

Example: in our running example q might be

- number of privacy ids in the dataset (scalar)
- number of rows in the dataset (scalar)
- number of restaurant visits per each day (vector, one coordinate per day).

Definition: Let **x**=(x_1,...x_n) be a real valued vector. Denote by

$$\begin{aligned} \|x\|_1 &= |x_1| + \ldots + |x_n| \\ \|x\|_2 &= \sqrt{x_1^2 + \ldots + x_n^2} \\ \|x\|_\infty &= \max\{|x_1|, \ldots, |x_n|\} \\ \|x\|_0 &= |\{i \in 1 \ldots n | x_i \neq 0\}| \end{aligned}$$

l_1, l_2, l_∞, l_0 norms[1] respectively..

Definition: Datasets **D, D**' are **neighboring** if they differ by data of 1 privacy id.

[1] Note that strictly speaking l_0 (number of non-zero coordinates) is not a vector norm, but it is convenient to use it and to refer to it uniformly with other norms

Definition: Let k be one of 0, 1, 2 or ∞. Then the **Lk-sensitivity** (i.e. L0, L1 etc) of the query **q** is

$$\Delta_k q = \max_{D, D' \text{neighbouring}} \|q(D) - q(D')\|_k$$

Remark: if **q** is a scalar query then L_1, L_2, L_∞-sensitivities are equal. But we need to be careful what scalar means. Eg. "number of restaurant visits per day" is not a scalar query, it returns a vector, namely a coordinate per day.

Definition: The **Laplace mechanism** (which is an ε-DP algorithm) is adding random noise to the query **q** according to the formula

$$Lap(D, q, \varepsilon) = q(D) + Lap(0, \frac{\Delta_1 q}{\varepsilon})$$

where Lap is a random sample from the Laplace distribution.

Definition: The **Gaussian mechanism** (which is $(\varepsilon, \boldsymbol{\delta})$-DP algorithm) is adding random noise to the query **q** according to the formula

$$Gauss(D, q, \varepsilon, \delta) = q(D) + N(0, \sigma^2(\varepsilon, \delta, \Delta_2 q))$$

where N is a random sample from the Normal distribution. The optimal value of sigma might be found by a numerical procedure described in this paper.

Let's consider some examples of computing sensitivity:

- number of unique privacy IDs in a dataset - L_0, L_1, L_2, L_∞-sensitivities are 1.
- number of restaurant visits - L_1, L_2, L_∞-sensitivities = maximum number of rows that correspond to a single privacy id. Without knowing/doing anything else, this is infinite. We can't add infinite noise. So we need to do contribution bounding to make this finite.

So in order to use the Laplace/Gaussian mechanism it is required to compute L_1, or L_2-sensitivities respectively. For convenience and simplicity, we typically reason about 0 and ∞ sensitivities instead, and use the following two formulas to bound L_1, and L_2-sensitivities.

$$s_1 \leq s_0 s_\infty$$

$$s_2 \leq \sqrt{s_0} s_\infty$$

Where s_0, s_1, s_2, s_∞ are L_0, L_1, L_2, L_∞ sensitivities of the query **q.** This approach is used in the following sections on cross-partition and per-partition contribution bounding.

Remark: these formulas follow from L_0, L_1, L_2, L_∞ norm properties.

Contribution bounding

As discussed in the previous section, the Laplace and Gaussian mechanisms require the underlying query to have known sensitivity (l_1 or l_2 respectively) and this can be achieved by bounding the l_0 and l_∞-sensitivities.

To bound the l_0-sensitivity by a positive integer s_0, we require that each privacy unit contributes to at most s_0 partitions. We call this **cross partition contribution bounding**.

To bound the l_∞-sensitivity by a positive number s_∞, we require that each privacy unit contributes at most s_∞ to each partition's underlying query (i.e. non-dp query). We call this **per-partition contribution bounding**.

These 2 types of contribution bonding will be considered in the following subsections.

Cross-partition contribution bounding

Let's recall that all collections (more details) considered further are generic collections that allow generic pipeline operations and each record has a privacy ID and a partition key.

Input: collection **col** with records (privacy_id, partition_key, value) and a parameter **max_partitions_contributed** (which will be the l_0-sensitivity**)**

For each privacy ID **pid:**

1. If records with privacy ID **pid** are associated with **max_partitions_contributed** partitions or fewer, then keep all these records.
2. Otherwise, choose **max_partitions_contributed** partitions among those associated with **pid**, and drop the rest.

We don't have to select which partitions to keep in a random way. It is possible to decide which contributions to keep using some other process, as long as this decision is done on a per individual basis, i.e., the data of individual A does not affect which data of individual B is kept. Doing so doesn't change **DP** guarantees.

Per-partition contribution bounding

To bound the contributions of each individual in each partition, there are multiple options. Below, we consider 2 of them.

Bounding each contribution

Input: collection **col**, int **max_contributions_per_partition** and **limits** on each input element (eg. **min/max** if we're aggregating numbers)

1. For each (privacy_id, partition_key) in **col** , keep at most **max_contributions_per_partition** elements and drop the rest.
2. Bound each element by given limits (eg. clipping to **[min, max]**).

This is a pretty universal way of bounding contributions per partition, it works for almost any metric (including all aggregations considered in this document).

Bounding aggregated contributions

Let's for example consider computing a sum per partition.

Input: collection **col** of elements (privacy_id, partition_key, double value), double **min_per_partition, max_per_partition**

1. Computes **sum** of values for each (privacy_id, partition_key) in **coll**
2. Clips the sum to **min_per_partition, max_per_partition**

Some other simple aggregations can also be bounded this way, though not all of them can.

Comparison of the two methods

Pros of bounding each contribution

1. It can be applied to elements of any nature, as long as we can specify sensitivity per element (e.g. numbers clipped to [min,max], vectors bounded by their l1, l2, l_∞ norm, etc.).
2. It gives better and more consistent results for compound metrics, like means and variances.

   Let's illustrate this by an example of computing the mean with the second method. To do so, we need to compute sum and count; in particular,we need to specify contribution bounds for both operations. How to do it in a consistent way? Suppose we chose some parameters, e.g.:

   for sum **min_sum_per_partition = 0, max_sum_per_partition = 5,**

   for count **min_count_per_partition = 0, max_count_per_partition = 3.**

   Let **data** be [1, 1, 1, 1, 1, 1, 1].

   Computing the sum and cliping to [**0, 5**] gives a result of 5. Computing the count and clipping it to [0,3] gives a result of 3.Thus, the result is 5/3, far from the true mean = 1, even before noise addition.

If we instead bound contributions using the first method we'll get better results. Imagine we select 3 contributions per partition, with a max_element_value = 5/3. Then the sum will be 3 and the count will be 3, yielding a correct mean of 1.

This example can be generalized to any possible parameters: one of the problems is that sum and count are computed on different sets of elements, which gives inconsistent results.

On the other hand, bounding each record avoids this type of bias, since the same subsampled elements are used for the sum and the count regardless of parameters.

Pros of bounding aggregated contributions

Bounding aggregated contribution can provide better utility, for example if values per partitions are {-1000, 1001} and min/max bounds are -5 and 5, the sum with method 2 will be **1** (which is the precise answer), while the individually-bounded elements will sum to 0.

Partition selection

Public and private partition keys

Definition: the partition keys for a DP algorithm are **public** if they are known in advance, at the start of the run of the DP algorithm. Otherwise, if the partition keys are not known in advance but are determined based on the data contributed by the individuals in the datasets, the partition keys are called **private**.

Remark: the same partition key might be considered public or private depending on the situation. More in examples below.

Examples:

1. In the restaurant visits example, imagine we want to compute the number of visits for a set of 50 restaurants for each day of a particular month. In this case there are 2 options for restaurant_id and date or (restaurant_id, date):
    a. consider them as public partition keys because we know in advance all of the possible dates and restaurants.
    b. consider them as private (by acting as if we don't know the list in advance).
2. If for the restaurant visits example we would like to compute the number of visits per restaurant per day for all restaurants and dates which are present in the dataset then (restaurant_id, date) are private partition keys.
3. If the task is to compute the frequency of words in a text and the partition keys are words. If not all words are known in advance, they are private partition keys. But if only a small subset of known words are of interest, they might be used as public partitions.

Public partitions should be part of the algorithm input and the result should contain all public keys and only them.

Example: Let's consider a query like *"Count of restaurant visits per year for 2016-2020 years"*. Then 2016, 2017 … 2020 are public keys. Assume the restaurant visits example dataset contains visits for 2014-2018 years. Then

- all visits from year 2014, 2015 should be dropped (they are not in public keys)
- years 2019, 2020 should be added to the result (which means noise needs to be added to these empty partitions in addition to the partitions present in the data).

Conversely, private partitions will contain only a subset of the partitions in the input dataset.

Public partitions

This section describes how public partition selection procedure works.

Let input be a collection **data** with elements (partition_key, values) and **public_partitions** is a collection with elements (partition_key). The output must contain all partitions in **public_partitions**, and only those: partitions missing from **data** must be present, and data points from **data** that are not in **public_partitions** should be dropped.

This can be done as two separate steps: first we drop partitions in the data that aren't in the public partitions, and then we add missing partitions from the public partitions. Adding missing public partitions depends on details of aggregations. That's why it will be considered later in the section.

To drop non-public partitions, we need to know for each **partition_key** from **data** whether **partition_key** is in **public_partitions.** There are 2 main ways to implement this.

With join

We can use the join operation of our pipeline framework to filter by **public_partitions.**

Input: collection **data** with elements (key, value) and **public_partitions** collection with elements (**key**)

Algorithm:

1. Join both collections per key.
2. Output all elements from **data** with keys from **public_partitions.**

In-memory

In case if **public_partitions** are small enough to fit in memory in some fast look-up structure (like a hash table) it is possible to do filtering of non-public partitions faster:

Algorithm: returns (partitions_key, values) pairs from **data** whenever partitions_key is in **public_partitions**.

When this algorithm is feasible, it is very fast, since it requires only one pass over the data. But if a public partition collection is large, the join method is the only option.

Private partition selection

In case when partition keys are not known in advance, they need to be selected in a DP fashion. Since all DP algorithms are randomized, any selection mechanism must be non-deterministic. The general intuition of these mechanisms is that the more individuals contribute to a partition, the larger the probability that the partition is in the result. This approach ensures that a single individual cannot significantly impact the presence of a particular partition key in the output.

There are multiple options:

1. Differentially private partition selection (a.k.a. optimal partition selection or truncated geometric thresholding)
2. Laplace/Gaussian thresholding

Remarks:

1. Implementing a single mechanism is enough to obtain the core functionality, but since some of them perform better than others depending on the input parameters, it might be worth implementing multiple mechanisms and automatically select the best one based on parameters.
2. Laplace thresholding never performs better (includes more partitions) than option 1 (“differentially private partition selection”).
3. Private partition selection mechanisms consume some of the privacy budget, and this must be accounted in the total privacy cost of the pipeline.
4. Private partition selection mechanisms also require contribution bounding, but the parameters for this contribution bounding do **not** need to be the same as the ones used for computing metrics.
5. Those 2 methods of partition selection are not the only possible ones. For example, the paper describes another class of approaches.

DP aggregation computations

This subsection describes how different differentially private aggregations are computed.

Computing different DP aggregations

We’re interested in the following DP aggregation functions:

1. COUNT

2. SUM
3. MEAN
4. VARIANCE
5. PRIVACY_ID_COUNT
6. COUNT_DISTINCT
7. PERCENTILE
8. MIN/MAX

where PRIVACY_ID_COUNT is the number of privacy identifiers per aggregation partition.

Computing COUNT, SUM, PRIVACY_ID_COUNT

These metrics might be computed with Laplace/Gaussian mechanisms with sensitivities computed by formulas. Note that COUNT might be considered as a sum of 1s.

Computing MEAN, VARIANCE

Non-DP mean and variance for numbers x_1, x_2, … x_n might be computed with formulas

$$Mean(X) = \frac{1}{n} \sum_{i=1}^{n} x_i$$
$$Var(X) = \frac{1}{n} \sum_{i=1}^{n} x_i^2 - (\frac{1}{n} \sum_{i=1}^{n} x_i)^2$$

That suggests the way to compute DP Mean and Var - compute count, sum and sum of squares with Laplace/Gaussian mechanisms with sensitivities computed by formulas.

Remarks:

1. Sometimes (n-1) is used instead of n in the denominator of variance. It doesn't change the way variance is anonymized.
2. There are modifications to minimize sensitivity (and improve utility) of computing mean and variance. For example if numbers are in [low, high], then the sum of normalized values x_1-mid, x_2-mid, …, x_n-mid where mid=(low+high)/2 has lower sensitivity and so it requires less noise than the sum of x_1, x_2, … x_n. Such modification doesn't change how conceptually both of these metrics are anonymized.

Computing COUNT_DISTINCT, PERCENTILE, ORDER STATISTICS (MIN/MAX etc.)

These metrics are pretty complicated and there are multiple approaches for each of these metrics with DP. The precise algorithms are outside of the scope of this document, but ideally we can use a well-tested library that supports distributed computation.

Performing aggregations

From a software engineering point of view it is convenient to have blackboxes (DP aggregators): objects that do aggregation and return anonymized results. These objects should be serializable since pipeline computations are typically running on distributed machines.

Aggregation of simple metrics

Let's consider the DP computation of multiple metrics (**Privacy Id Count**, **Count**, **Sum**, **Mean**, **Variance**).

As it is shown above these metrics can be computed by summing the corresponding numeric values and applying Laplace/Gaussian mechanism. Note that there's overlap between the metrics - Count, Mean, and Variance all use a count, and Sum, Mean, and Variance all use a sum. If we're computing more than one of these metrics, we can compute each value once and re-use them to compute the desired metrics.

Remarks:

1. This approach is conceptually equivalent to summing vectors with coordinates that correspond to unique id count, count, sum and sum of squares. From the design point of view and for the code readability, using a black-box DP aggregator object for doing that looks preferable.
2. To compare the budget spent on each "coordinate", different mechanisms might be used - incl. precise formulas for Gaussian composition, or simple/advance composition for Laplace. Details are outside of the scope of this doc.

Comparison of computing simple metrics with one aggregator vs computing separately.

Regardless of whether metrics are computed together or separately, we need to compute specific sums and noise them with the Laplace or Gaussian mechanism.

The difference is that computing metrics together allows more effective budget usage and more consistent results when computing more than 1 metric simultaneously. For example if the goal is to compute count and mean, then computing metrics together allows us to compute the count only once and to reuse it for the mean computation.

Adding missing public partitions

As noted in the public partition section, we need to ensure that public partitions which are missing in the input data are present in the output. This section describes how to do this.

There are 2 questions:

1. How do we do a DP aggregation on an empty input set?

2. How do we ensure that we do that for every partition in the public set that's missing from the input?

Let's consider how to deal with those questions.

DP aggregation results for empty partitions

The results for empty partitions should be ε-indistinguishable from partitions with a single input (i.e. neighboring databases) according to corresponding DP guarantees.

What to return for an empty partition depends on the metric we're calculating? Let's consider it for metrics in which we are interested:

1. COUNT, SUM, PRIVACY_ID_COUNT, COUNT_DISTINCT: output is **0 + noise**.
2. MEAN, VARIANCE: If mean and variance are calculated as post processing on a series of DP sums and counts (i.e. count, sum, sum of squares), then we can use **0 + noise** for those values and do the post-processing as usual..
3. MIN/MAX, PERCENTILE is outside of the scope of this doc. We suggested earlier that these be computed with a library; a good library should include appropriate behavior for empty input sets.

Remark: The behavior for empty partitions should be encapsulated in DP aggregators if they're implemented as black boxes.

Note that we don't need a separate noising process for empty partitions - we can insert the appropriate empty value into the input (discussed in the next section), and proceed with the computation. Noise will be added by the same mechanisms that handle the non-empty partitions.

Note also that adding an extra copy of the empty value to an aggregation won't (and shouldn't) change the result.

Adding results for empty partitions to the output

Let input be a collection of **data** with elements (partition_key, value) and **public_partitions** is a collection with elements (partition_key).

There are multiple ways to add missing partitions, depending on whether public partitions are computed with a join or in memory.

1. Public partitions with join: **public_partitions** is joined with **data** on partition_key. We can use the pipeline frameworks to do a full join, i.e. process all the keys from both collections. Then if there are no values for a key, we can insert the empty value for that aggregation (discussed in the previous subsection).
2. In-Memory public partitions: for every key in **public_partitions**, we add (key, empty_value) to the **data**.

Privacy budget tracking

We want to have a way of calculating and specifying the total privacy loss budget for an entire pipeline. We can use a number of different techniques for this (e.g. basic composition, privacy loss distributions, Renyi differential privacy, etc) without changing the way the pipeline works. In a DP pipeline, the budget is consumed by:

1. Noise addition
2. Partition selection (in case of non-public partitions).

Further discussions of budget tracking is outside the scope of this doc.

Inputs of the DP algorithm

We can now see the pipeline will need the following inputs:

Collection (each element contains a privacy key and a partition key) and **params.** Where **params** are**:**

1. Total privacy loss budget - (eps, delta)
2. Specification of metrics to compute (e.g. metric name, fields).
3. Portion of the budget allocated to each metric (or it might be chosen automatically)
4. Parameters for contribution bounding.
5. Public partitions (if present).
6. Noise type - Laplace of Gaussian (or it might be chosen automatically).

Computation graph

A DP pipeline computation might be represented in the form of a **computation graph** - directed acyclic graphs, such that each node is a simple pipeline operation like map, join, combine etc. For convenience we can name sub-graphs of computational graphs as high level operations - like contribution bounding, partition selection etc, keeping in mind that each of those operations is a computation graph consisting of simple operations.

Also conditional nodes will be used. Conditions will depend on metaparements, e.g. “if public partition”. In the DP aggregations we consider the computation graph does not depend on the results of an aggregation function; it will only depend on aggregation metaparameters (like presence or absence of public partitions, etc).

The goal of this section is to show computation graphs for computing DP aggregations per partition on data with privacy ids.

Discussion on order of different stages

There are some limitations on the order:

1. Per-partition contribution bounding should happen before aggregation .
2. Cross partition contribution bounding should happen before aggregation.
3. Aggregation should be before adding noise.

The most flexible is partition selection. As discussed below, the order in which partition selection and contribution bounding are performed, impacts performance and the utility of the final output.

Contribution bounding - Partition selection (CB-PS)

It means that cross partition contribution bounding is performed for all partitions, some of those partitions might be dropped during partition selection.

Partition selection - Contribution bounding (PS-CB)

It means that cross-partition contribution bounding is only applied to those partitions which will be present in the final result. So this is more “data friendly” than CB-PS.

As discussed above, private partition selection algorithms require a limit on the number of user contributions. So in this case we would need to bound cross-partition contributions and perform partition selection. Then, returning to the original, un-bounded input, we would drop partitions that were not selected and perform cross-partition contribution bounding again (followed by per-partition contribution bounding and aggregation). Having a second contribution bounding has a performance cost (especially when you consider that we can otherwise compute the aggregation and partition selection results for each partition at the same time, saving a join), but it might make sense if keeping the maximum number of partitions is important.

For public partitions there is no need to do an extra round of contribution bounding, but dropping partitions before aggregating is still more expensive when a join is used.

Comparison of order of partition selection

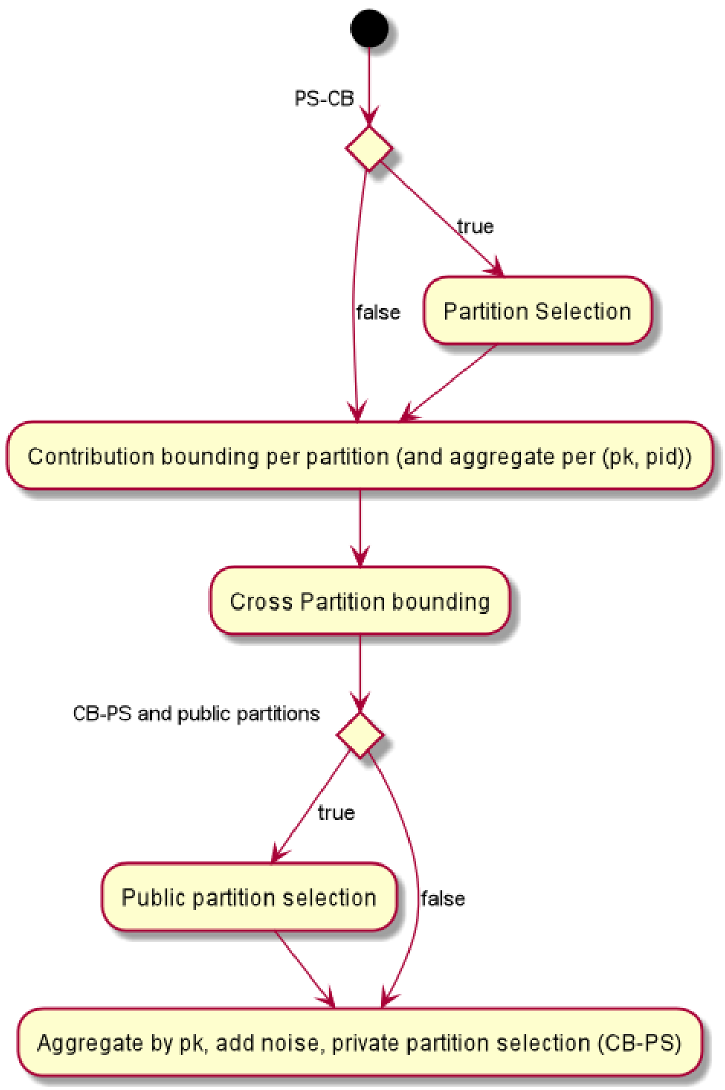
1. PS-CB keeps more data, but (excluding in-memory public partition) it requires dropping unselected partitions from the input data rather than the aggregated results. The input data is usually much larger, and therefore this is more expensive. It may also require an extra contribution bounding step..
2. Partition selection by a randomized procedure in CB-PS order is cheaper:
    a. For private partitions no need to join with selected partition keys, because it’s possible to compute count of individuals per partition at the same time we’re aggregating metrics.
    b. For public partitions we can join public partition keys with the aggregated results rather than the input, and aggregated data is usually much smaller than raw data.

Computation graphs for aggregation by partition key with privacy ids

In the graphs, the following abbreviations will be used

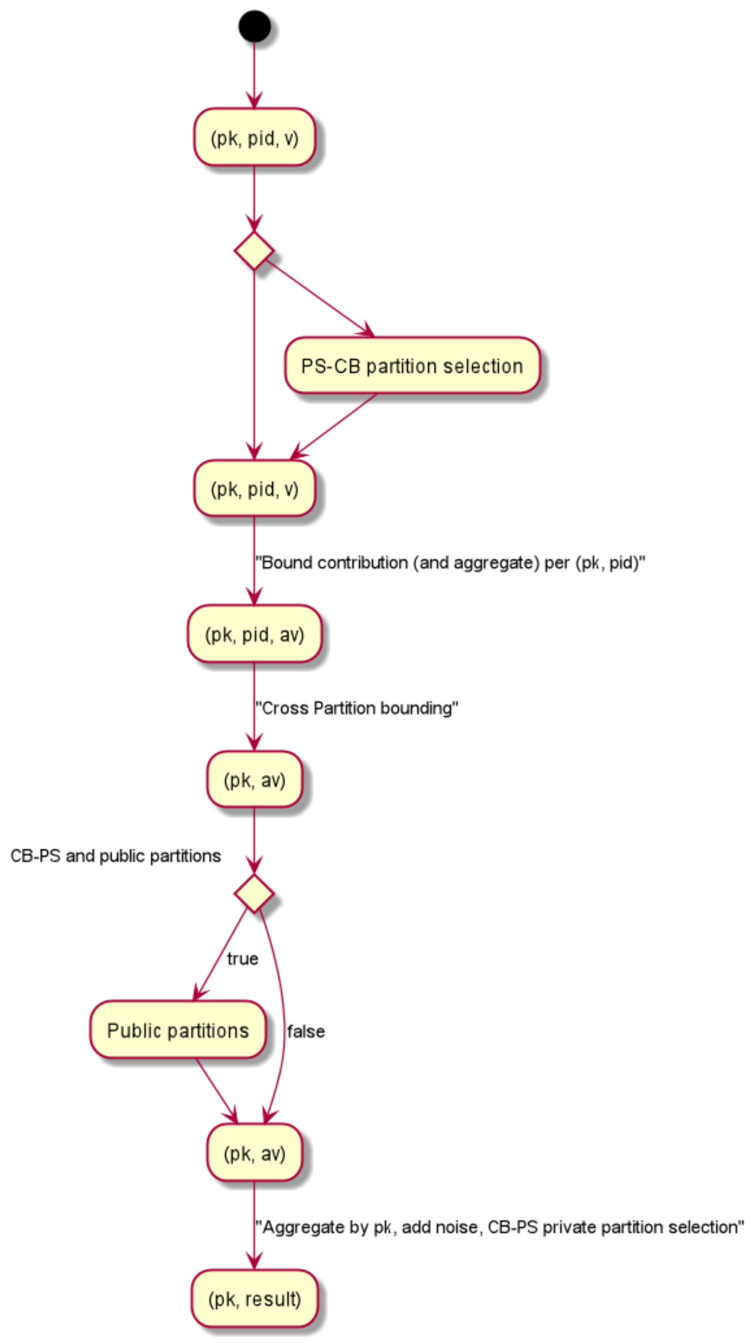
1. pid - privacy id
2. pk - partition key
3. v - value
4. av - aggregated per privacy id value
5. aav - aggregate per (privacy id, partition key) value
6. PS-CB - partition selection-contribution bounding
7. CB-PS - contribution bounding-partition selection

The computational graph is the following:


PS-CB
true
false
Partition Selection
Contribution bounding per partition (and aggregate per (pk, pid))
Cross Partition bounding
CB-PS and public partitions
true
false
Public partition selection
Aggregate by pk, add noise, private partition selection (CB-PS)

Computation graph

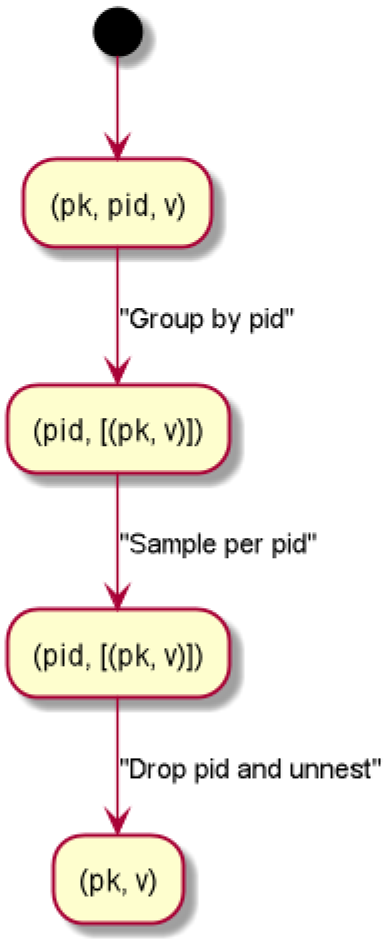
For simplifying and clarifying implementation it makes sense to supplement this diagram with types of elements. Which would introduce some additional steps which are required for transforming types.



(pk, pid, v)
PS-CB partition selection
(pk, pid, v)
"Bound contribution (and aggregate) per (pk, pid)"
(pk, pid, av)
"Cross Partition bounding"
(pk, av)
CB-PS and public partitions
true
Public partitions
false
(pk, av)
"Aggregate by pk, add noise, CB-PS private partition selection"
(pk, result)

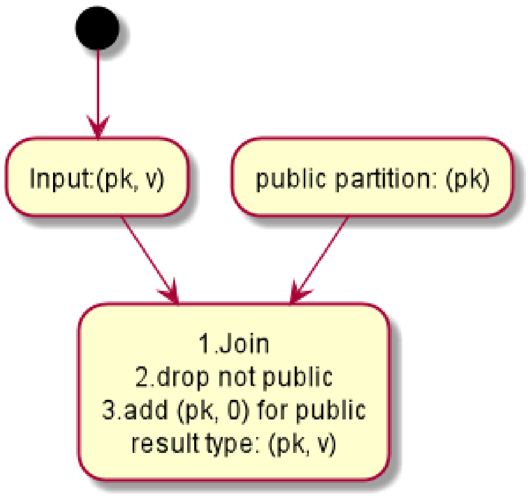
Computation graph with element types

Cross partitions contribution bounding (this diagram assumes that it's after per-partition contribution bounds):

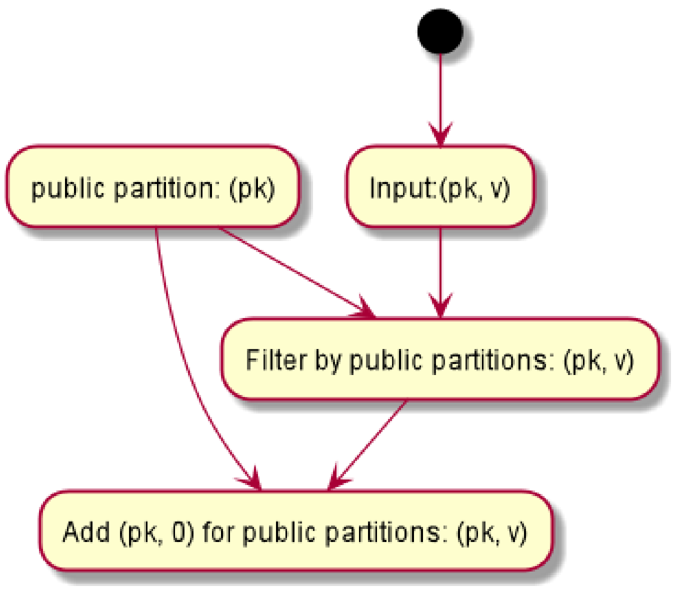


Cross partition contribution bounding

There are 2 options for public partitions:



Partition selection with join


public partition: (pk)
Input:(pk, v)
Filter by public partitions: (pk, v)
Add (pk, 0) for public partitions: (pk, v)

In-memory partition selection