SORTaki: A Framework to Integrate Sorting with Differential Private Histogramming Algorithms

Stylianos Doudalis
School Of Information & Compute Sciences
University of California, Irvine, USA
sdoudali@ics.uci.edu

Sharad Mehrotra
School Of Information & Compute Sciences
University of California, Irvine, USA
sharad@ics.uci.edu

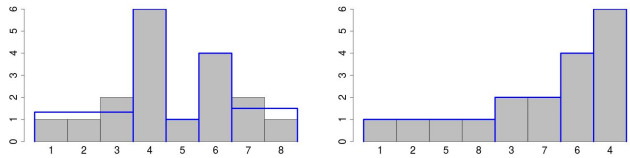
***Abstract*—Differential privacy has been established as the primary framework for privacy preserving data-sharing. In the context of query answering through histograms, most of the data-dependent solutions are composed of two steps: a partitioning phase that splits the histogram into bins and a finalizing step that approximates each bin with its average frequency or other similar statistics. Solutions that sort the histograms' values prior to the partitioning phase can improve the utility of the final output. In this paper, we build SORTaki, a framework that integrates sorting with any partitioning and finalizing mechanism. Using SORTaki, we modify existing partitioning and finalizing solutions, as well as propose new ones, that mitigate the error of the final approximation up to 70% over existing sorting or non-sorting based algorithms. Additionally, we perform a principled and thorough empirical evaluation of current and proposed techniques, that highlights the right settings to use sorting and when to avoid it.**

***Index Terms*—databases, privacy**

I. Introduction

Today, personal information is being collected through various channels, such as, online surveys, electronic medical records, web search history, online transaction history etc. Sharing such microdata (individual specific data) can be of great scientific and commercial value at the cost of privacy violations. One powerful notion of privacy that has emerged to address these concerns is that of differential privacy (DP), first proposed by Dwork et al. [4]. The DP criterion states that the output of a differentially private algorithm should not change significantly depending upon whether a single user's data are included in the input dataset or not. In other words, a DP-compliant algorithm should be "insensitive" to the presence or absence of a user's records in the input dataset. Given a user query, and a privacy budget, Differential Privacy perturbs the real answer by adding noise.

Histograms are one of the most convenient and easily understood ways of representing a distribution and they are widely used for data analysis and data mining tasks. DP histogram publishing techniques can be separated into data-dependent [1], [2], [8], [19], [20] and data-independent [3], [7], [17] based on whether or not they make decisions that are influenced by the form of the distribution. Like traditional histogramming techniques [13], data-dependent algorithms consist of two phases. They begin by partitioning the domain into uniform bins and they represent each bin by the average frequency (or other similar metrics, e.g. median). The partitioning is based on a error measure that quantifies a bin's "goodness". Most of the times the error introduced due to a bin can be decomposed into two parts, the *generalization error* and the *perturbation error*. The former is the error created by grouping together values that have different real frequencies, while the latter is due to the noise added by differential privacy. The two types of errors have opposite behavior. If we create a single bin, we have the biggest possible generalization error, but we minimize the perturbation error. On the other extreme, if we assign a single bin per histogram value, then the generalization error is zero, but the perturbation error is maximized. It is this trade-off that all data dependent DP histogramming techniques exploit, in order to create better approximations of the underlying distribution compared to data independent techniques.



(a) Original (b) Sort based

Fig. 1: Sorting could potentially help to create bigger and with lower generalization error bins.

There are a variety of data-dependent algorithms that experiment with different partitioning algorithms. The partitioning algorithm may try to optimize histogram error based on L_1 or L_2^2 norm, it might approximate a bin's frequency using the average or median frequency. The partitioning algorithm might decide the bins' boundaries by working on noisy frequencies [3], [18], [19] or it might directly output them [1], [2], [19]. All this prior work has a common goal, that is to find a better way of quantifying a bin's error, in order to detect good regions of uniformity. In all cases the output is the same, a set of non-overlapping contiguous bins.

In [8], [20] the authors explore a new direction for improving the quality of data-dependent histogramming techniques. They observe that in the context of differential privacy we can go beyond contiguous bins. The continuous bins make sense,

This material is based upon work supported by DARPA under grants FA8750-16-20021.

when the histogram acts as the summary of the underlying distribution. However, under differential privacy, while compactness of the outputted bins is a desirable property, the main focus is to reduce the error of the final noisy approximation. For example, consider the distribution in Fig. 1a that contains eight values. The overimposed blue line represents the histogram created by some data-dependent algorithm. Ideally, the algorithm should output uniform bins with zero generalization error. But, if we limit ourselves only to contiguous bins, then we will never consider clustering together values 1,2,5 and 8 because they are not "close" in the original domain. If we sort the values based on their frequency, i.e. Fig. 1b, then we could potentially create uniform bins. On a high level in [8], [20], sorting works as follows. A part of the budget is spent in order to learn the frequency of each value in the histogram. The noisy frequencies are used to re-order to domain in decreasing order. The partitioning and finalizing step are applied on the "sorted" domain that then the values are re-arranged to their original order.

In this paper, we continue and generalize the work started by [20]. Specifically, we make the following **contributions**: 1) We built SORTaki, a general framework for incorporating sorting to any existing or future histograming technique. SORTaki has four basic components: 1) An *Initializer*, which collects some initial statistics about the underlying distribution. 2) a *Partitioner*, which represents any logic that can partition a domain into a set of contiguous bins, 3) a *Finalizer*, which given a set of non-contiguous bins, creates the final approximation for each bin and 4) a *Sorter* which can re-order bins based on some criterion. SORTaki provides a flexible framework that we can use to easily experiment with various components and integrate sorting to existing histogramming algorithms like DPCube [18] and DAWA [2]. 2) We create a new partitioner and finalizer. In [20] a portion of the budget is used to decide the "sort" order and perform the partitioning. The information learned during the previous step is discarded and the final approximation is created using only the remainder of the privacy budget. In section V, we first create a new *finalizer*, which can leverage the information collected during every step. Based on the new *finalizer*, we also derive a new bin error measure, which takes into account 1) the new *finalizer* and 2) the fact the generalization error is computed using noisy data. Our experiments show that the new *finalizer* and bin error measure can reduce the error up to 45%, for high levels of privacy budget or small domain sizes. Additionally, inspired by the partitioner presented in DAWA [2], we create a scalable dynamic programming based partitioner that offers up to 70 % improvement over the greedy partitioner presented in AHP [20]. 3) We perform a principled and thorough evaluation of old and new components based on the principles presented in DPBench [6], a general framework for evaluating differential private algorithms. Our goal is not only to evaluate the performance of existing and new techniques, but also to create general guidelines for using sorting based mechanisms, that could be potentially integrated into meta-algorithms like Pythia [9]. Based on our experiments the "ideal" distribution for sorting has a very big domain with hundred of thousand values, a very big scale (number records), the shape of distribution is not smooth, e.g. a power low distribution, and the workload consists of small range queries. Sorting best works in large domains, because it is easier to form big non-contiguous bins. Large queries is a bad setting, because sorting creates non-contiguous bins that don't respect the ordering in the original domain. Therefore the bigger the range query is, the more likely is that it will be the union of multiple bins in the presence of sorting. Sorting actually tries to apply the *partitioner* and *finalizer* over a "smoothed" version of the distribution. Therefore, if the latter is already partially of fully "sorted", then what sorting tries to achieve comes for free. The smaller the scale of a dataset is the more likely is that there are big "regions" of zero counts in the distribution, which can be detected by any non-sorting based techniques.

The remainder of the paper is organized as follows. Section II contains the principles of differential privacy and section III the related work. Section IV describes SORTaki framework. Section V has the improved versions of partitioner and finalizer. Section VI describes our experimental setup, followed by the empirical results in section VII. The paper ends with the conclusion in section VIII.

II. Preliminaries

In this section, we give the definition of differential privacy [4] and describe its fundamental properties. Differential privacy is based on the notion of neighbor databases.

Definition II.1 (Neighbor Databases [4])**.** Two databases D and D' are neighbors if they differ by at most one record. That is $|(D \setminus D') \cup (D' \setminus D)| = 1$. We denote the neighbors of D as $N(D)$.

Definition II.2 (Differential privacy [4])**.** Let $\mathcal{T}$ be a domain of databases. A mechanism $\mathcal{M} : \mathcal{T} \to \mathcal{O}$ satisfies ϵ-differential privacy if for all pairs of neighboring databases D and D' and all $S \subseteq \mathcal{O}$

$$Pr[M(D) \in S] \leq e^{\epsilon} Pr[M(D') \in S].$$

Differential privacy has two important properties - sequential and parallel composition [12]. Sequential composition says that the privacy loss composes sensibly with repeated applications of differentially private algorithms.

Theorem II.1 (Sequential Composition)**.** *Let $\mathcal{M}_1, \mathcal{M}_2, \cdots \mathcal{M}_k$ be a set of mechanisms such that $\mathcal{M}_i$ satisfies ϵ_i-differential privacy. Then $\mathcal{M}_1 \circ \mathcal{M}_2 \circ \cdots \circ \mathcal{M}_k$ satisfies $\sum_{i=1}^{k} \epsilon_i$-differential privacy.*

In certain situations, composing differentially private mechanisms does not cause a degradation in privacy. This is true when the mechanisms operate on disjoint parts of the domain.

Theorem II.2 (Parallel Composition)**.** *Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ be disjoint subsets of the domain. That is $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$, and $\mathcal{T}_i \subseteq \mathcal{T}$. Let D be a database, and $\mathcal{M}_1, \ldots, \mathcal{M}_k$ be $\epsilon_1, \ldots, \epsilon_k$*

differentially privacy algorithms respectively. The sequence $\mathcal{M}_i(D \cap \mathcal{T}_i)$ *satisfies* $\max_i \epsilon_i$*-differential privacy.*

The standard mechanism for releasing the output of continuous function is Laplace mechanism, that adds noise proportional to the function's global sensitivity:

Definition II.3 (Global Sensitivity [4])**.** Let $f : \mathcal{D} \rightarrow \mathbb{R}$ be a function and D, D' be two neighbor databases, the global sensitivity of f is $\triangle f = \max_{D,D'} \|f(D) - f(D')\|_1$.

Laplace mechanism is defined as

Definition II.4 (Laplace mechanism [4])**.** Let $f : \mathcal{D} \rightarrow \mathbb{R}$ be a function, D be a database and $Lap(\beta)$ be a sample for Laplace distribution with mean 0 and scale β. Mechanism $M(D) = f(D) + Lap(\triangle f/\epsilon)$, satisfies ϵ-differential privacy.

III. Related work

Publishing histograms in the context of differential privacy is a well studied problem. In this section we classify the majority of algorithms based on a variety of criteria.

DP histogramming techniques can be classified as data dependent vs data independent. The major advantage of **data independent** [1], [3], [7], [14], [17] techniques is their stability, as they have nice foreknown behavior. Stability, however, comes at the cost of performing poorly in case of small range queries and not exploiting the possible inherent uniformity that might exists in the dataset. All **data dependent** algorithms are two phase. The algorithms start with a bin construction phase, which partitions the domain into disjoint bins, followed by a querying phase that learns the frequencies of each bin. Different approaches have been proposed for both phases. In [1], [19] the authors publish the bins using the exponential mechanism. In [19] the authors create the bins based on noisy frequencies using a dynamic programing approach, while in [3], [18] they employ a Kd-tree approach for partitioning multidimensional data. In [2], the binning algorithm selects the optimal scheme by adding noise directly to candidate bin's error (generalization plus perturbation error). The usual approach for a second phase is to query the bin's average (or median) frequency. Instead, in [19] the authors use the boost technique [7], which improved the algorithm's stability and its performance on range queries.

DP histogramming methods can also be classified based on additional knowledge used by the algorithm. In [2], [5], [11] the authors present **workload aware** histogramming techniques using both data dependent and data independent approaches. Also in [5] the algorithm can be classified as following the interactive setting where the user is not necessarily interested in the complete distribution. Orthogonal to the work of publishing histograms is that of learning **k-way marginal contingency** tables [15].

Sorting has been explored in [8], [20]. In [8] the authors exploit sorting for the specific scenario of publishing histograms when the database contains multiple records of a user. They tackle the problem by publishing a histogram with fixed size bins. Their approach exploits part of the privacy budget to sample the data distribution, in order to sort the domain based on frequencies and to learn the optimal bin size. In [8] the authors show, that the resulting data dependent approach outperforms the data independent strategy of publishing a DP histogram with fixed size bins in the original domain order. While the approach in [8] is related to us in motivation, our goals are different. Instead of the specific scenario explored in [8], we explore a general mechanism to incorporate sorting into any data dependent algorithm and investigate, in depth, the conditions under which such an approach may improve utility. Our work can be considered as a continuation and generalization of the work started in [20].

Given the plethora of the proposed algorithms and the fact that there is no single best solution, a new line of work [6], [9] tries to impose order in the "chaos". In [6], the authors present DPBench, a framework for evaluating differiential private histogramming solutions. Our experimental setup in section VI is based on the principles of DPBench. In [9], Ios et at. created a meta-algorithm, which enables the user to select the right histogramming mechanism. The conclusions from on our empirical evaluation can help improve such meta-algorithm on how to use sorting based techniques.

IV. SORTaki framework

In order to extend data dependent algorithms with sorting, we first describe an abstract/conceptual view of such algorithms which will allow us to incorporate different strategies of sorting into the existing algorithms.

A. Abstract Components

While individual details differ, we can intuitively view all data-dependent algorithms as consisting of three major components – an *initializer* that might create a noisy representation of the original histogram, a *partitioner* that splits the domain into a set of not overlapping bins, and a *finalizer*, that, given the bins, creates the final histogram. The privacy budget is split between the three components, in the ration γ_{in}, γ_p and $1 - \gamma_{in} - \gamma_p$, where the overall privacy budget is ϵ. We first describe the three components in more detail and then show how this abstraction can be instantiated to realize two popular DP algorithms – DAWA [2] and DPCube [18].

Initializer. The *initializer* optionally prepares the original histogram $\mathcal{H}$ for the *partitioner* module. For example, DPCube's *partitioner* makes the decisions over noisy frequencies, while DAWA's *partitioner* requires access to the real data. A basic version of the *initializer* is described in algorithm 1. If γ_{in} is greater than zero, e.g. DPCube, then the module will output a noisy version of $\mathcal{H}$, using the privacy budget $\gamma_{in}\epsilon$. Otherwise, it returns an empty array. Argument $\mathcal{T}$ represents a set of extra parameter that might be unique to custom *initalizers*. Thus, we propose the following template:

$$\widetilde{\mathcal{H}} = Initializer(\mathcal{H}, \epsilon, \gamma_{in}, \mathcal{T})$$

Partitioner. The *partitioner* is a wrapper around the logic that splits the original domain into contiguous bins. The input to the *partitioner* is a set of bins $\mathcal{V}$. Initially, all the bins inside

Algorithm 1: Basic Initializer: $\mathcal{H}, \epsilon, \gamma_{in}, \mathcal{T}$

$\widetilde{\mathcal{H}} = H, \epsilon_{in} = \gamma_{in}\epsilon$
if $\gamma_{in} = 0$ **then return** [];
return $(\tilde{f}_{1,\epsilon_{in}}, \dots \tilde{f}_{|H|,\epsilon_{in}})$, s.t. $\tilde{f}_{i,\epsilon_{in}} = f_i + Lap(1/\epsilon_{in})$

$\mathcal{V}$ have size equal to 1. Using the information contained in $\mathcal{H}$ or $\widetilde{\mathcal{H}}$, the *partitioner* decides how to merge the smaller bins to larger ones. The binning process spends γ_p portion of the total budget ϵ. Usually, the *partitioner* needs to make assumptions about how the *initializer* and the *finalizer* work. Therefore, it requires to know their portion of the budget, i.e. γ_{in} and γ_p. Finally, $\mathcal{T}$ represents a set of parameters that are unique to every *partitioner* algorithm, such as thresholds or parameter controlling the behavior of internal data structures, e.g. the fanout of a tree. The previous description is summarized into the following template:

$$\mathcal{V} = Partitioner(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$$

Finalizer. The finalizer creates the final differentially private histogram $\widetilde{\mathcal{H}_f}$ using the remainder of the budget, i.e. $\gamma_f\epsilon$, and $\widetilde{\mathcal{H}}$. Given a set of bins, a *finalizer* employees a query strategy and it also leverages post processing techniques, e.g. least square fit, in order to improve the utility. The query strategy, i.e. the set of queries asked using γ_f, might be pre-defined, e.g. asking a single query per bin, or workload aware. The *finalizer* has similar template to the *partitioner*.

$$\widetilde{\mathcal{H}_f} = Finalizer(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$$

Mapping existing algorithms to the abstraction layer. Using the *initializer*, the *partitioner* and the *finalizer* modules, it is straightforward to describe data-dependent histogramming algorithms as presented in algorithm 2. The initializer spends budget ϵ_{in} to generate a noisy histogram using Laplace mechanism. $\mathcal{V}$ is initialized into a set of bins that contains a single value per bin. The *partitioner* decides the final bins and the finalizer creates the output histogram $\widetilde{\mathcal{H}_f}$.

Algorithm 2: $\mathcal{H}, \epsilon, Initializer, \gamma_{in}, Partitioner, \gamma_p, Finalizer$

$\widetilde{\mathcal{H}} = Initializer(\mathcal{H}, \gamma_{in}, \mathcal{T})$
$\mathcal{V} = Identity(|H|)$
$\mathcal{V} = Partitioner(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$
return $Finalizer(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$

Next, we illustrate how the above abstract formulation of data-dependent algorithms can be instantiated to realize two well known algorithms –viz, DAWA [2] and DPCube [18].

DAWA. DAWA does not contain an explicit initializer, i.e. $\gamma_{in} = 0$. Instead, DAWA's *partioner* decides the bins while looking at the real data. It considers all possible contiguous bins as possible candidates. For each bin, it calculates its error based on the real data and the error is perturbed. The noise addition guarantees that the bin selection is a differentially private process. A deterministic dynamic programming algorithm selects the best set of non overlapping contiguous bins based on the noisy bins' errors. DAWA's *partitioner* has an optional parameter that allows to select only candidate bins of size multiple of 2. DAWA's *finalizer* employs a tree based query strategy. For example imagine that $\mathcal{H}$ was split into two bins v_1 and v_2. The query strategy consists of 3 candidate query, one for each bin and one for for their union. For each query it allocates a privacy budget based on a workload. For example, if the workload consists of small queries, then it will decide to ask a single query per bin. On the other extreme, if the workload contains a lot of large ranges, it will favor queries that span multiple bins. The extra parameters $\mathcal{T}$ for the finalizer consist of the workload and the fanout of the tree.

Symbol	Explanation
$N(b)$	Random variable sampled from Laplace distribution with scale b.
v / $\|v\|$ / $\mathcal{V}$	Bin / Bin's size / Set of bins
γ_{in} / γ_p / γ_f	Portion of budget used by initializer / partitioners / finalizer.
ϵ	Total privacy budget.
ϵ_{in} / ϵ_p / ϵ_f	$\gamma_{in}\epsilon$ / $\gamma_p\epsilon$ / $\gamma_f\epsilon$
f_i / $\tilde{f}_{i,\epsilon}$	Ith value's real frequency / $f_i + N_i(1/\epsilon)$
$\mathcal{H}$ / $\widetilde{\mathcal{H}}$ / $\widetilde{\mathcal{H}_f}$	original histogram / initializer's output / finalizer's output.
$\mathcal{T}$	Optional module parameters.
$Identity(\|\mathcal{H}\|)$	Creates one bin per value
avg_v / $\widetilde{avg}_{v,\epsilon}$	v's real average frequency / $avg_v + N_v(\frac{1}{\|v\|\epsilon})$
$\widetilde{avg}_v$	v's final average noisy frequency.
$\mathbb{E}[X]$	Expected value of random variable X
$\mathbb{V}[X]$	Variance of random variable X
$EE(v)$	v's expected error
$UEE(v)$	v's unbiased expected error

DPCube. DPCube starts by creating a noisy version of $\mathcal{H}$, i.e. $\gamma_{in} > 0$. The partitioner does not spend additional budget, i.e. $\gamma_p = 0$. DPCube's *partitioner* uses a KD-Tree like algorithm to split the domain into bins. A tree node is split, if its noisy variance is bigger than a threshold ξ. The finalizer asks a single query (average) for each bin, followed by a post-processing step that combines the average stats with the information inside $\widetilde{\mathcal{H}}$.

B. Adding Sorting

In the previous subsection, we saw how techniques like DAWA and DPCube can be implemented in our framework. In this subsection, we extend algorithm 2 to support sorting based solution and we use AHP [20] as an example.

Bin as a subset of the domain. Given that sorting does not respect the original domain order, we extend the definition of bin, v, to a subset of the domain. Going back to the introductory example in figure 1a, every bin corresponds to a single range (or contiguous bin). On the other hand, after sorting (figure 1b), a bin can be any subset of the original domain. The complete partitioning, $\mathcal{V}$, is constrained such that: (a) no two bins overlap, and (b) union of the bins covers the

entire domain. We will, henceforth, refer to such generalized bins, simply as bins.

Sorter. A *sorter* takes as input a set of bins $\mathcal{V}$ and a noisy histogram $\widetilde{\mathcal{H}}$ and generates as output the input bins sorted based on some criterion, e.g. the average bin frequency.

$$\mathcal{V} = Sorter(\mathcal{V}, \widetilde{\mathcal{H}})$$

Algorithm 3: Input: $\mathcal{H}, \epsilon, Initializer, \gamma_{in} > 0$, $Partitioner, \gamma_p, Finalizer$

$\widetilde{\mathcal{H}} = Initializer(\mathcal{H}, \gamma_{in}, \mathcal{T})$
$\mathcal{V} = Identity(|H|)$
$\mathcal{V} = Sorter(\mathcal{V}, \widetilde{\mathcal{H}})$
$\mathcal{V} = Partitioner(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$
$\widetilde{\mathcal{H}_f} = Finalizer(\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p, \gamma_f, \mathcal{T})$
return $\widetilde{\mathcal{H}_f}$

The complete logic of a data-dependent algorithm enhanced with sorting is presented in algorithm 3. The main change is that the partitioner is preceded by the *sorter*, that reorders the bins. The *partitioner* and *finalizer* work on the "sorted" domain the same way that they would in the original domain, but the *finalizer* is additionally required to create final approximation based on the original domain order. Since sorting has to be done in a differental private fashion, it is always performed on noisy frequencies, thus parameter γ_{in} is required to be greater than 0.

AHP. AHP [20] can be easily implemented based on algorithm 3. AHP's spends part of the budget in order to learn a noisy version of the distribution like in the basic intitializer (algorithm 1), but additionally is employs a post-processing step which map every frequency smaller that a threshold ξ to zero. We implemented the thresholding step as an extension of the basic initializer. After sorting the domain, AHP created bins based on $\widetilde{\mathcal{H}}$ without spending extra budget, i.e. $\gamma_p = 0$. AHP's finalizer consists of asking for every bin's average frequency using the remainder budget.

Above, we showed how AHP algo can be implemented using our framework. In a similar manner, algorithsm such as DAWA and DPCube can be augmented to exploit sorting as well, by adding a step between initialization and partitioning to sort the domain. Indeed, we implemented several of these algorithms and compare them with and without sorting in section VII.

V. New Components

In the previous section, we presented our framework and we showed how DAWA, DPCube and AHP can be instantiated. In this section, we suggest new components based on the shortcomings of previous work or the new ideas applied in the literature in different settings.

Our first contribution is a new *finalizer* . AHP's *finalizer* is simple. Each bin's frequency is approximated by the average frequency learned using the remainder of privacy budget, i.e. $\gamma_f \epsilon$ and any information present in $\widetilde{\mathcal{H}}$ is just ignored. In this section, we create a new *finalizer* that combines the counts in $\widetilde{\mathcal{H}}$ with the average learned with the last part of the privacy budget. DPCube has a similar *finalizer*, that combines the two sources of information with a least square technique, but we provide a closed form solution. We go one step more and we create a bin error formula that takes into account the new type of *finalizer*. Our second extension are an adjustable greedy and dynamic programming based *partitioners* that can partition the domain using any bin error formula.

A. Finalizer and Bin Error Estimation formulas

The goal of this subsection is to derive 1) the close form formula used by the *finalizer* and 2) the new bin error formula.

Calculate a bin's final frequency. Imagine that the *partitioner* somehow decided the final bins $\mathcal{V}$. What should be the final frequency for each value inside $v \in \mathcal{V}$?

$$\widetilde{avg}_v = \alpha_1 \frac{1}{|v|} \sum_{i \in v} \tilde{f}_{i,\epsilon_{in}} + \alpha_2 \cdot \widetilde{avg}_{v,\epsilon_f}$$

For that task we use the bin's noisy average frequency, i.e. $\widetilde{avg}_v$, which is the combination of two estimators. The first one computes the average using $\widetilde{\mathcal{H}}$, while the second one is learned from the database using the remainder of the budget. Next we derive weights, α_1 and α_2, such that $\widetilde{avg}_v$ is unbiased and it has minimum variance. Let N_i and N_v be Laplace random variables with scales $1/\epsilon_{in}$ and $1/(|v|\epsilon_f)$ respectively. The estimator's bias is

$$\mathbb{E}[\widetilde{avg}_v - avg_v] = 0 \Leftrightarrow (\alpha_1 + \alpha_2 - 1) avg_v = 0 \tag{1}$$

The estimator's variance is

$$\mathbb{E}\left[(\widetilde{avg}_v - avg_v)^2\right] \overset{(1)}{=\!=} \mathbb{E}\left[\left(\frac{\alpha_1}{|v|}\sum_{i=1}^{|v|} N_i + \alpha_2 N_v\right)^2\right]$$

$$= \frac{\alpha_1^2}{|v|^2}\sum_{i=1}^{|v|}\mathbb{E}\left[N_i^2\right] + \alpha_2^2\mathbb{E}\left[N_v^2\right] + \frac{2\alpha_1\alpha_2}{|v|}\mathbb{E}\left[N_v\right]\sum_{i=1}^{|v|}\mathbb{E}\left[N_i\right]$$

$$+ 2\alpha_1^2\sum_{i=1}^{|v|}\sum_{j=i+1}^{|v|}\mathbb{E}[N_i]\mathbb{E}[N_j] = \frac{\alpha_1^2}{|v|^2}\sum_{i=1}^{|v|}\mathbb{V}(N_i) + \alpha_2^2\mathbb{V}(N_v)$$

$$= \frac{\alpha_1^2}{|v|^2}\sum_{i=1}^{|v|}\frac{2}{\epsilon_{in}^2} + \alpha_2^2\frac{2}{|v|^2\epsilon_f^2} = \frac{2\alpha_1^2}{|v|\epsilon_{in}^2} + \frac{2(1-\alpha_1)^2}{|v|^2\epsilon_f^2} \tag{2}$$

The minimum of the previous quadratic equation is obtained when

$$\alpha_1 = \frac{\epsilon_{in}^2}{\epsilon_{in}^2 + |v|\epsilon_f^2} \text{ and } \alpha_2 = \frac{|v|\epsilon_f^2}{\epsilon_{in}^2 + |v|\epsilon_f^2} \tag{3}$$

Equation 3 says that the bigger ϵ_f is and the bigger the bin is, the more accurate is $\widetilde{avg}_{v,\epsilon_f}$ as an estimator of the bin's average frequency. At this point we have the closed form formula for the new finalizer.

Evaluate a bin's quality, over the real data. The next question we need to answer is how good of an approximation is $\widetilde{avg}_v$ for v? We measure the quality with the expected sum

of squared differences between f_i and $\widetilde{avg}_v$ or $EE(v)$ for short.

$$EE(v) = \mathbb{E}\left[\sum_{i=1}^{|v|}(f_i - \widetilde{avg}_v)^2\right] = \sum_{i=1}^{|v|}\mathbb{E}\left[(f_i - \widetilde{avg}_v)^2\right] =$$
$$= \sum_{i=1}^{|v|}\mathbb{E}\big[(f_i - avg_v + avg_v - \widetilde{avg}_v)^2\big]$$
$$= \sum_{i=1}^{|v|}(f_i - avg_v)^2 + \sum_{i=1}^{|v|}\mathbb{E}\left[(avg_v - \widetilde{avg}_v)^2\right]$$
$$+ \sum_{i=1}^{|v|}2(f_i - avg_v)\mathbb{E}\left[(avg_v - \widetilde{avg}_v)\right]$$
$$\underline{\underline{(2),(3)}} \quad \sum_{i=1}^{|v|}(f_i - avg_v)^2 + \frac{2}{\epsilon_{in}^2 + |v|\epsilon_f^2} \tag{4}$$

According to equation (4), a bin's error can be decomposed into two terms - the generalization error, i.e. $\sum_{i=1}^{|v|}(f_i - avg_v)^2$, and noise variance, i.e. $\frac{2}{\epsilon_{in}^2+|v|\epsilon_f^2}$. The generalization error depends on the uniformity of the bin, with higher uniformity resulting in smaller error. The noise variance depends on the bin's size, the total budget and the budget assignment. Specifically, the bigger the bin is, the smaller the variance is and less affected it is by parameters ϵ_{in} and ϵ_f. We get the worst case noise variance when ϵ_{in} and ϵ_f are equal.

Evaluate a bin's quality, in the presence of noisy data. Equation (4) allows us to calculate a bin's error when we have access to the real data. An indirect *partitioner* requires a formula that can uses over noisy data, i.e. $\widetilde{\mathcal{H}}$. A simple solution is to compute the generalization error with the noisy versions of f_i and avg_v, i.e. $\widetilde{f}_{i,\epsilon_{in}}$ and $\widetilde{avg}_{v,\epsilon_{in}}$. We have to make sure that the generalization error is adjusted by a bias term such that estimator of a bin's error using $\widetilde{\mathcal{H}}$ remains an unbiased process. In the following derivation N_i and N_j are Laplace random variables with scale $1/\epsilon_{in}$.

$$Bias =$$
$$= \mathbb{E}\left[\sum_{i=1}^{|v|}(\widetilde{f}_{i,\epsilon_{in}} - \widetilde{avg}_{v,\epsilon_{in}})^2\right] - \sum_{i=1}^{|v|}(f_i - avg_v)^2$$
$$= \sum_{i=1}^{|v|}\mathbb{E}\left[\left(N_i - \sum_{j=1}^{|v|}N_j/|v|\right)^2\right]$$
$$= \sum_{i=1}^{|v|}\left\{\mathbb{E}\left[N_i^2\right] + \frac{\sum_{j=1}^{|v|}\mathbb{E}\left[N_j^2\right]}{|v|^2} - \frac{2}{|v|}\sum_{j=1}^{|v|}\mathbb{E}\left[N_iN_j\right]\right\}$$
$$= \sum_{i=1}^{|v|}\left\{\frac{2}{\epsilon_{in}^2} + \frac{1}{|v|}\frac{2}{\epsilon_{in}^2} - \frac{2}{|v|}\frac{2}{\epsilon_{in}^2}\right\} = (|v|-1)\frac{2}{\epsilon_{in}^2}$$

Using the noisy frequencies and the bias term equation (4) becomes

$$UEE(v) = \sum_{i=1}^{|v|}(\widetilde{f}_{i,\epsilon_{in}} - \widetilde{avg}_{v,\epsilon_{in}})^2 - (|v|-1)\frac{2}{\epsilon_{in}^2} + \frac{2}{\epsilon_{in}^2 + |v|\epsilon_f^2}$$

The bias term encourages the creation of bigger bins. As the privacy budget ϵ_{in} decreases, the bias term increases and bigger bins are favored. In [19], the authors showed how to decompose the sum of squares into the sum of the squared frequencies and the square of the sum of the frequencies.

$$UEE(v)^1 = \sum_{i=1}^{|v|}\widetilde{f}_{i,\epsilon_{in}}^2 - \frac{\left(\sum_{i=1}^{|v|}\widetilde{f}_{i,\epsilon_{in}}\right)^2}{|v|} - (|v|-1)\frac{2}{\epsilon_{in}^2} + \frac{2}{\epsilon_{in}^2 + |v|\epsilon_f^2} \tag{5}$$

Thus we get the optimized formula (5) which can be computed in linear time, $O(|v|)$.

B. Modules

In this subsection we present the new *finalizer* and the generic *partitioners* based on the formulas derived in previous subsections.

Algorithm 4: WAF($\mathcal{V}, \mathcal{H}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p = 0, \gamma_f, \mathcal{T} = \{\}$)

```
H̃_f = [ ], ε_in = γ_in ε, ε_f = γ_f ε
α_1 = ε_in^2 / (ε_in^2 + |v| ε_f^2), α_2 = 1 − α_1
foreach v ∈ V do
    avg̃_v = α_1 (1/|v|) Σ_{i∈v} f̃_{i,ε_in} + α_2 avg̃_{v,ε_f}
    for i ∈ v do H̃_f[i] = avg̃_v
return H̃_f
```

Weighted Average Finalizer, WAF. Algorithm 4, contains the logic of the finalizer component. As described in the first part of the previous subsection, for each bin v, the *finalizer* approximates the frequency of every value in v, with the weighted average of the frequencies in $\widetilde{\mathcal{H}}$ and $\widetilde{avg}_{v,\epsilon_f}$. WAF does not require any extra parameters. Compared to WAF, AHP's [20] *finalizer* uses only $\widetilde{avg}_{v,\epsilon_f}$.

Algorithm 5: GP($\mathcal{V}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p = 0, \gamma_f, \mathcal{T} = \{EF\}$)

```
V' = {}, ε_in = γ_in ε, ε_f = γ_f ε, v = V[1]
foreach i = 2 ∈ up to |V| do
    v' = v ∪ V[i], er_1 = EF(v', ε_in, ε_f)
    er_2 = EF(v, ε_in, ε_f) + EF(V[i], ε_in, ε_f)
    if er_1 < er_2 then v = v' else V' = V' ∪ v, v = V[i]
return V' ∪ v
```

Generic Partitioners. In [20] the authors presented a greedy and a dynamic programming based *partitioner*. In this subsection, we port the *partitioners* to our framework and we generalize them, such that they can work with any bin error formula. The greedy *partitioner* is presented in pseudocode in algorithm 5. Since all the decisions are made over $\widetilde{\mathcal{H}}$, the *partitioner* spends no extra privacy budget, i.e. γ_p is zero. The generalization happens through the extra parameter EF which

[1] For the sake of simplicity, equations EE (4) and UEE (5) are presented to have a single parameter, i.e. v. In reality, they have three input parameters, v, ϵ_{in}, and ϵ_f.

represents the error formula, that is used to estimate a bin's quality. EF has three argument a bins v, the privacy budget used during the initialization step, ϵ_{in}, and the remainder of the budget ϵ_f. During every iteration, lines 3 to 4, the algorithm decides whether or not to merge the current bin v with the next one $\mathcal{V}[i]$, by simply comparing the error of the combined bins with the error of keeping them separated.

Algorithm 6: DPP($\mathcal{V}, \widetilde{\mathcal{H}}, \epsilon, \gamma_{in}, \gamma_p = 0, \gamma_f, \mathcal{T} = \{EF\}$)

1 $\mathcal{S}$ = [,], $\epsilon_{in} = \gamma_{in}\epsilon$, $\epsilon_f = \gamma_f\epsilon$
2 **foreach** $e = 1$ *up to* $|\mathcal{V}|$ **do**
3 **foreach** $s = e$ *down to* 1 **do**
4 $\mathcal{S}$[s,e] = $EF(\mathcal{V}[s:e], \epsilon_{in}, \epsilon_f)$
5 $\mathcal{S}$[s,e] = min($\mathcal{S}$[s,e], $\min_{s \le i < e}$ ($\mathcal{S}$[s,i]+$\mathcal{S}$[i+1,e]))
6 **return** CreatePartitioning($\mathcal{S}$)

Algorithm 6 contains pseudocode for the dynamic programming *partitioner*, that has the same signature as 5. From line 1 to 2 the algorithm iterates over all possible ranges and finds the best solution for each range. In this case, the range is over the bins inside $\mathcal{V}$ not over the actual histogram values. For example $EF(\mathcal{V}[s:e], \ldots)$ means to compute the error by merging all the bins in $\mathcal{V}$ starting from position s and ending at position e. In line 6, function *CreatePartitioning* used the information inside $\mathcal{S}$, to trace the best partitioning $\mathcal{V}$. Inspired from [2], we implemented a version of *partitioner*, that considers only bins whose size is multiple of 2, thus reducing the complexity of the algorithm to $O(n \log^2 n)$, and making a better alternative of the greedy *partitioner* for large domains.

In this section, we saw new *partitioners* and *finalizers*. In the next two sections we continue with the experimental evaluation.

VI. Experimental Setup

In this section, we present our experimental framework. In [6], Hay et al. present DPBench, a framework for evaluating differential private algorithms. They identify three basic properties for each dataset, 1) the domain size, 2) the scale, i.e. the number of records, and 3) the shape of the distribution. In order to create a fair setup for comparing various datasets, they employ a **data generator**, which allows to create synthetic datasets that have the same domain size and scale, but different shapes. In addition, they prove that most of the existing histogramming techniques follow the principle of **scale-epsilon exchangeability**. If an algorithm is scale-epsilon exchangeable, decreasing the privacy budget ϵ by a factor α has the same effect on the performance of the algorithm as decreasing the scale by the same factor. Scale-epsilon exchangeability significantly reduces the number of experimentation configurations that we have to consider, because it creates a direct connection between the privacy budget and the scale.

TABLE I: Dataset properties

Dataset	Domain	Scale	# bins based on ϵ		
			1	0.5	0.1
NetTrace	1024	100K	65	37	19
Wage-Per-Hour	1024	100K	431	261	144
Amazon	500K	7.5M	311K	235K	95K
Bid-IP	500K	7.5M	266K	167K	38K
Bid-Freq	500K	7.5M	154K	58K	4K
Ipums-Inc	48K	1M	16K	11K	5K
MD-Sal-Orig	99081	136K	6K	4K	1K
MD-Sal	48K	1M	30K	19K	5K

Table I contains a summary of the **datasets**. For each dataset, we report the domain size, the scale and the number of bins over the real data for three levels of privacy budget ϵ. The number of bins was calculated using AHP's partitioner for $\gamma = 0.5$, but any other partitioner could have been used. The number of bins offers some intuition on the shape of the underlying distribution, as it provides a sense on the uniformity of the dataset. Uniformity is what any histogramming technique attempts to leverage and, as explained in the introduction, sorting tries to create uniformity when it does not exist. If the underlying distribution is completely uniform or close to a power law distribution, e.g. NetTrace and Bid-Freq, then the number of bins is small and sorting would offer limited benefit. On the hand, if a distribution is "random", i.e. there is big variability between the frequencies of neighbor values, then it would have little uniformity and the number of bins will be large. In that case algorithms that consider only contiguous bins would have low performance and sorting can offer the biggest benefit. Next we go over the datasets used in our experiments. **Wage-Per-Hour** was extracted from the 1994 and 1995 population surveys conducted by the U.S. Census Bureau[1]. The original distribution is the marginal of the wage per hour. **NetTrace** [7] was derived from an IP-level network traced collected at a major university. In the original dataset, each record reports the number of external hosts connected to an internal host. The histogram is the number of records per number of external hosts. From *Wage-Per-Hour* and *NetTrace* we isolated the first 1024 values and generated a histogram with scale equal to 100K. **Amazon** dataset was collected by SNAP group [10]. It contains metadata and review information about 548,552 different products (e.g. Books, music CDs, DVDs and VHS video tapes)[2]. The distribution is the number of reviews per product. The **Bid** datasets are derived from a Kaggle competition[3] launched by Facebook, which aims to detect bots in online auctions. Both distributions are the number of bids per IP address. In case of Bid-IP the values are ordered alphabetically by the IP address, while in case Bid-Freq the values are ordered in decreasing frequency. From the Amazon and Bid dataset we isolated the first 500,000 values and we use the data generator in order to generate a scale of 7.5 Million. **IPUMS-Inc** is the distribution of the pre-tax income

[1] http://archive.ics.uci.edu/ml/datasets/Census+Income
[2] http://snap.stanford.edu/data/amazon-meta.html
[3] https://www.kaggle.com/c/facebook-recruiting-iv-human-or-bot

TABLE II: Acronyms overview.

Algorithm	Components
Identity	Laplace mechanism with scale $1/\epsilon$
AHP	AHP's *finalizer* and bin error formula
WAF	*finalizer* and bin error formula presented in section V-A
[G | D]X	[Greedy | Dynamic Programming] *partitioner* + algorithm X
DPC	DPCube*partitioner* + WAF *finalizer*
DAWA	Regular DAWA [2]
sDAWA	DAWA's *partitioner* + WAF *finalizer*
sX	Sorting + algorithm X
tX	Frequency thresholding + algorithm X

or losses from 2001 to 2011 as recorded by IPUMS [16]. **MD-Sal** datasets are based on the Maryland salary database of 2012 [4]. *MD-Sal-Orig* is the original histogram that contains a single bin per salary. From the IPUMS and MD-Sal datasets we kept the first 48,000 values and generated two datasets with 1 million records.

Error metric. For our experiments we use the scaled average per query error (definition VI.1) presented in [6]. The new error metric is the average over a workload of queries normalized by the scale of the distribution.

Definition VI.1 (Scaled Average Per Query Error)**.** Let $\mathbf{x}$ be a data vector with scale $s = \|\mathbf{x}\|_1$ and $\mathbf{W}$ be a workload of q queries with real answers $\mathbf{y}$. Let $\hat{\mathbf{y}} = \mathcal{K}(x, \mathbf{W}, \epsilon)$ be the noisy output of algorithm $\mathcal{K}$. Given an error metric L, the scaled average per query error is defined as $\frac{1}{sq} L(\hat{\mathbf{y}}, \mathbf{y})$.

Workloads. In DPBench, the authors use the prefix workload which consists of all range queries starting from the first value. The prefix workload offers a measure of the performance over all possible query sizes. In order to fully analyze the performance of sorting, we separate the type of workload based on the type of attribute. In case of **categorical** variables, we use the identity workload that consists of all range queries of size 1. When we evaluation **continous** attributes, we differentiate over two types of workloads consisting of small or big range queries. The **small range** workload consists of all possible queries of size 1 to 10, while the **big range** workload consists of all possible queries of range between 100 and 1000 in increments of 100.

VII. Experimental Results

Using the experimental framework presented in the previous section, we present the right settings to use sorting and when to avoid it. Our experiments show that the **"ideal" scenario** for a sorting based technique consists of 1) a workload composed of small range queries and 2) a dataset that has a very large domain, a large scale and a shape that is "random" and not partially or fully "sorted". Exponential or power law distributions are examples of "sorted" distributions.

All the empirical results are presented in terms of the scaled average per query error. In case of tabular representation of the results, the reader is advised to pay attention to the **error scale** present at the bottom of the table. Multiplying the value in a cell by the error scale yields the actual scaled average per query error. Before we go over the results, in Table II, we overview the experimental configurations and their **acronyms**. In all the experiments, the Identity mechanism, that publishes all frequencies with Laplace Mechanism, serves as a baseline. AHP and WAF refer to the *finalizer* and the bin error estimation formula used by the *partitioners*. When we prefix AHP or WAF with the letter 'G' ('D'), then the best partitioning is calculated using the greedy (dynamic programming) based *partitioner* presented in section V-B. When a configuration is prefixed with letter 's', then it is enhanced with sorting. When an algorithm is preceded by letter 't', then the basic *initializer* is replaced by *initalizer* with the extra thresholding step, that set "small" frequencies to 0, as designed in AHP. Note that cofiguration **'stGAHP'** describes the complete AHP algorithm as presented in [20]. DAWA refers to the original algorithm as described in [2]. We did not implement DAWA, instead we extended the authors' code. We chose DAWA, because in [6], it is shown to have the most stable behavior when compared with other 12 techniques. The sorting enhanced version of DAWA, i.e. sDAWA, uses the DAWA's *partitioner*, but the WAF *finalizer*, because it can leverage the information learned during the initialization step. Also the privacy budget for running DAWA's *partitioner* is set to $0.25 \cdot (1 - \gamma_{in})$. DPC and sDPC configurations use DPCube's *partitioner* and the WAF *finalizer*, because WAF is more efficient to use compared to the least square solution proposed in [18]. DPCube's *partitioner* presented as interesting alternative to our dynamic programming alternative. Additionally, by setting threshold parameter ξ to $\frac{2}{\epsilon^2}$, the *partioner*'s performance improved considerably.

The remainder of the section is organized as follows. In subsection VII-A, we examine the behavior in case of large categorical attributes. In subsection VII-B, we present the results for large domain continuous datasets over small and large range queries. Finally, in subsection VII-C, we show the performance over small domain sizes.

A. Large, categorical domains

In this subsection, we examine the behavior of sorting for large categorical domains. In Table III, we primarily focus on how sensitive sorting is to parameter γ_{in}, the portion of the budget used by the *initializer*. In Table IV, we vary the shape of the distribution and the privacy budget ϵ. Recall that based on scale-epsilon exchangeability, decreasing ϵ by 10 times has the same effect as reducing the scale by the same factor.

The main conclusion from Table III is that our proposed dynamic programming based *partitioner* reduces the error considerably over the greedy *partitioner* as shown by comparing lines sGAHP and sDAHP. Not only does sDAHP have smaller error than sGAHP, but also it wins over the baseline identity approach for $\epsilon = 1$. Note that when ϵ is equal to 1, only sorting based algorithms (excluding the greedy) win over the baseline solution. The optimal choice for parameter γ_{in} depends on the type of *partitioner*. In case of DAWA, DPCube and dynamic programming based *partitioner* the error is minimized when γ_{in} is 0.9 , while for the greedy *partitioner* is 0.7 .

[4] http://data.baltimoresun.com/salaries/state/cy2012/export-for-sun-users.csv

TABLE III: Large categorical. Amazon dataset. Error vs ϵ and γ_{in}. Bold is minimum error per row and ϵ.

ϵ	1.0				0.1			
γ_{in}	0.3	0.5	0.7	0.9	0.3	0.5	0.7	0.9
Identity	0.27	0.27	0.27	0.27	26.67	26.67	26.67	26.67
GAHP	**0.61**	0.79	1.44	6.35	34.26	**32.56**	33.68	99.41
sGAHP	1.67	0.82	**0.79**	2.89	54.05	28.72	**23.18**	51.03
stGAHP	1.67	0.82	**0.79**	2.89	54.46	28.59	**23.18**	50.80
sDAHP	1.59	0.67	0.37	**0.24**	52.99	25.82	15.85	**10.91**
sDWAF	1.61	0.67	0.37	**0.24**	53.43	26.00	15.90	**11.26**
DPC	0.70	0.55	0.46	**0.33**	**30.59**	36.77	39.17	30.68
sDPC	1.60	0.67	0.37	**0.24**	53.31	25.92	15.88	**11.36**
DAWA	0.80	0.80	0.80	0.80	30.93	30.93	30.93	30.93
sDAWA	0.93	0.60	0.38	**0.28**	31.33	25.76	19.73	**18.03**
	error scale : 10^{-6}							

TABLE IV: Large categorical. Error vs shape and ϵ. $\gamma_{in} = 0.9$. Cases in bold is the minimum over the column.

Dataset	Amazon			Bid-IP			Bid-Freq		
ϵ	1.0	0.5	0.1	1.0	0.5	0.1	1.0	0.5	0.1
Identity	0.27	1.07	26.67	0.27	1.07	26.67	0.27	1.07	26.67
sDAHP	0.24	0.82	**10.91**	0.27	0.80	**6.44**	0.25	0.72	5.83
sDWAF	**0.24**	**0.82**	11.26	**0.26**	**0.78**	6.50	**0.24**	**0.70**	6.06
DPC	0.33	1.31	30.68	0.34	1.31	29.85	0.33	1.26	29.16
sDPC	0.24	0.83	11.36	0.26	0.78	6.55	0.24	0.70	6.13
DAWA	0.80	2.68	30.93	0.85	2.21	11.83	0.47	0.71	**1.51**
sDAWA	0.28	1.03	18.03	0.33	1.10	15.14	0.31	1.04	13.93
	error scale : 10^{-6}								

In Table IV, we examine the performance as a function of the shape and privacy budget ϵ. We only present the results for the best algorithms and parameter γ_{in} is fixed to 0.9 . In general, we observed that for very large domains γ_{in} equal to 0.9 offers the smallest error independently of the shape and ϵ. The main observation from Table IV is how the shape of the distribution can influence the final performance. When the distribution is "random", e.g. Amazon and Bid-IP, sorting based techniques win by far. On the other hand, when the distribution is "sorted", e.g. Bid-Freq, then sorting is not always the answer. For large values of privacy budget ϵ sorting wins, while for smaller values of ϵ a non-sorting based technique (DAWA) has the smallest error. For large domains, we did not observe a big difference between choosing AHP's or WAF's *finalizer* and bin error formula. The reason is that when the final bins are big, then the best estimator for a bins frequency is the average learned by spending the remainder portion of the budget γ_f. In practice, WAF *finalizer* defaults to AHP's *finalizer*. Yet we observed that WAF (AHP) is slightly better for bigger (smaller) values of ϵ.

B. Large, continuous domains

In the previous subsection, we show how sorting can benefit in case of large categorical domains. In this subsection, we change the domain type to continuous attributes. We examine small and large range workload separately.

In Table V, we fix the dataset to MD-Sal-Orig and we observe the behavior of the configurations, that we saw in the previous section, as a function of parameter γ_{in}. The performance is evaluated over the small range workload. The conclusions are similar to the ones in case of categorical attributes and the identity workload. Sorting when combined with the dynamic programming or the DPCube partitioner can greatly improve the performance for small size range queries.

TABLE V: Large continuous. Dataset MD-Sal-Orig. Error vs ϵ and γ_{in}. Small range workload. Bold is minimum error per row and ϵ.

ϵ	1.0				0.5			
γ_{in}	0.3	0.5	0.7	0.9	0.3	0.5	0.7	0.9
Identity	0.81	0.81	0.81	0.81	3.24	3.24	3.24	3.24
GAHP	0.93	0.85	**0.43**	0.86	3.76	3.28	**1.34**	2.14
sGAHP	0.62	0.32	**0.25**	0.42	1.68	0.86	**0.61**	0.94
stGAHP	0.62	0.33	**0.24**	0.43	1.76	0.87	**0.62**	0.93
sDAHP	0.60	0.30	0.19	**0.14**	1.65	0.79	0.51	**0.38**
sDWAF	0.64	0.30	0.19	**0.18**	1.68	0.80	**0.51**	0.55
DPC	**0.69**	1.06	1.23	0.98	**2.68**	4.17	4.90	3.91
sDPC	0.61	0.31	0.19	**0.18**	1.73	0.81	**0.52**	0.59
DAWA	0.26	0.26	0.26	0.26	0.77	0.77	0.77	0.77
sDAWA	0.39	0.39	**0.36**	0.38	1.03	1.01	**1.00**	1.14
	error scale : 10^{-4}							

TABLE VI: Large continuous. Error vs shape and ϵ. Small range workload. $\gamma = 0.9$. Bold is minimum over column.

Dataset	MD-SAL			IMPUS-INC		
ϵ	1.0	0.5	0.1	1.0	0.5	0.1
Identity	**0.11**	0.44	11.00	0.11	0.44	11.00
sDAHP	0.13	0.41	**3.63**	0.08	0.27	**2.64**
sDWAF	0.12	**0.39**	4.08	**0.06**	**0.22**	3.00
DPC	0.13	0.53	13.23	0.13	0.54	13.43
sDPC	0.12	0.40	4.40	0.07	0.22	3.32
DAWA	0.20	0.59	5.37	0.11	0.33	4.08
sDAWA	0.15	0.54	8.79	0.18	0.48	9.81
	error scale : 10^{-4}					

In Table VI, we fix parameter γ_{in} and we vary the shape of the distribution. The experimental results show that sorting helps in case of small range queries. The WA finalizer offers an advantage for higher levels of privacy budget and AHP's finalizer for smaller levels. In case of the MD-SAL dataset and privacy budget 1.0 the Identity baseline algorithm remains the best solution. The previous results come to verify the fact that there is no algorithm that offers the best solution over all settings and meta algorithms like Pythia [9] are useful in order to solve the problem of algorithm selection. Having said that, sorting based algorithms still offer performance very close to the baseline.

In Table VII, we keep the experimental setting the same as with table VI, but we evaluate the performance in terms of large range queries. In this setting sorting does not help or it does not help in a consistent fashion. DAWA without sorting offers the best utility. We identified two reasons. Firstly, in case of large range queries, DAWA's *partitioner* is superior to the frequency based *partitioners*, e.g. DPCube, AHP or WAF. Secondly, sorting works by grouping together values that in the original domain could be "very far". When a range query is small, then it is answered using a single or a small number of bins. On the other hand, a large range query will be answered by "combining" multiple bins. Without sorting a big range query will be answered by only a limited number of bins.

In this subsection, we examined the effect of sorting for continuous attributes. The basic conclusion is that sorting helps for small range queries. In case of large range queries, non-sorting based techniques like DAWA will offer better utility.

TABLE VII: Large continuous. Error vs shape and ϵ. Big range workload. $\gamma_{in} = 0.9$. Bold is the minimum over column.

Dataset	MD-SAL			IMPUS-INC		
ϵ	1.0	0.5	0.1	1.0	0.5	0.1
Identity	**0.11**	0.44	10.97	0.11	0.44	10.97
sDAHP	0.14	0.59	24.87	0.45	2.37	20.46
sDWAF	0.12	0.44	15.91	0.30	1.63	14.16
DPC	0.13	0.57	14.56	0.13	0.55	13.06
sDPC	0.13	0.42	15.90	0.28	1.42	12.19
DAWA	0.11	**0.31**	**2.55**	**0.07**	**0.19**	**2.46**
sDAWA	0.30	1.11	67.98	1.03	5.27	172.15
	error scale : 10^{-2}					

TABLE VIII: Small continuous. Dataset Wage-Per-Hour. Error vs ϵ and γ_{in}. Small range workload. Bold is minimum error per row and ϵ.

ϵ	1.0				0.5			
γ_{in}	0.3	0.5	0.7	0.9	0.3	0.5	0.7	0.9
Identity	1.10	1.10	1.10	1.10	4.39	4.39	4.39	4.39
GAHP	**1.59**	2.04	2.47	6.45	**5.67**	5.95	6.05	23.16
sGAHP	2.83	1.66	**1.60**	4.66	7.19	**3.73**	3.77	7.35
stGAHP	3.99	2.68	**1.98**	3.56	10.70	5.57	**4.39**	7.33
sDAHP	2.48	1.47	**1.14**	1.85	5.93	3.25	**2.77**	4.95
sDWAF	2.66	1.46	1.10	**1.01**	7.66	3.55	**3.00**	3.71
DPC	**1.21**	1.62	1.80	1.35	**4.25**	6.64	6.80	5.44
sDPC	2.57	1.52	1.17	**1.12**	6.50	3.64	**3.03**	4.33
DAWA	1.18	1.18	1.18	1.18	3.09	3.09	3.09	3.09
sDAWA	2.54	2.16	**1.72**	1.73	5.97	5.97	**5.45**	5.58
	error scale : 10^{-4}							

C. Small domains

For the last set of experiments we focus on small domain continuous datasets. In tables VIII and IX, we examine the effect of parameter γ_{in} and the shape of the distribution in case of the small range workload.

As the domain size decreases, the "difficulty" of the dataset increases. With the average bin size being small, the performance of sorting based techniques is more sensitive to the right choice parameter γ_{in} as a function of privacy budget ϵ. The smaller ϵ is, the best performance is achieved for a smaller value of γ_{in}. Additionally the gap between WAF and AHP *finalizer* increases, with WAF being considerably better (worse) than AHP's *finalizer* when the privacy budget is big (small). Table IX re-affirms that when the shape of the distribution is "sorted" then DAWA has better performance. On the hand, the more "random" a distribution is the the more beneficial it is to choose a sorting based solution.

VIII. Conclusion and Future Work

In this paper, we introduced SORTaki a framework for incorporating sorting with any existing or future data dependent histogramming technique. We developed a new *finalizer* and proposed a scalable dynamic programming based *partitioner*, that further improve sorting based techniques over the state of the art. We performed a thorough and principled evaluation of sorting enhanced algorithms. Our experiments showed that sorting can help the most in case of small workloads and datasets with large domains, big scales and "ramdon" shapes. In order to make sorting an out of the box solution, we envision that the right path would be to incorporate the conclusions of our experiments into meta-algorithms like Pythia [9].

TABLE IX: Small continuous. Error vs shape and ϵ. Small range workload. $\gamma_{in} = 0.9$. Bold is minimum over column.

Dataset	WAGE-PER-HOUR			NETTRACE		
ϵ	1.0	0.5	0.1	1.0	0.5	0.1
Identity	1.10	4.39	109.84	1.10	4.39	109.84
sDAHP	1.85	4.95	**39.83**	1.69	3.95	93.39
sDWAF	**1.01**	3.71	82.05	0.89	3.36	75.26
DPC	1.35	5.44	132.36	1.43	5.07	141.11
sDPC	1.12	4.33	87.68	0.89	3.60	85.64
DAWA	1.18	**3.09**	63.45	**0.18**	**0.38**	**6.26**
sDAWA	7.35	12.96	122.89	2.66	18.66	133.68
	error scale : 10^{-4}					

References

[1] G. Acs, C. Castelluccia, and R. Chen. Differentially private histogram publishing through lossy compression. In *ICDM*, pages 1–10, 2012.

[2] G. M. Chao Li, Michael Hay and Y. Wang. A data- and workload-aware query answering algorithm for range queries under differential privacy. pages 341–352, 2014.

[3] G. Cormode, C. Procopiuc, D. Srivastava, E. Shen, and T. Yu. Differentially private spatial decompositions. ICDE '12, pages 20–31.

[4] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. TCC'06, pages 265–284, 2006.

[5] M. Hardt and G. N. Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *FOCS*, pages 61–70. IEEE, 2010.

[6] M. Hay, A. Machanavajjhala, G. Miklau, Y. Chen, and D. Zhang. Principled evaluation of differentially private algorithms using dpbench. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 139–154, New York, NY, USA, 2016. ACM.

[7] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, pages 1021–1032, 2010.

[8] G. Kellaris and S. Papadopoulos. Practical differential privacy via grouping and smoothing. PVLDB'13, 2013.

[9] I. Kotsogiannis, A. Machanavajjhala, M. Hay, and G. Miklau. Pythia: Data dependent differentially private algorithm selection. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1323–1337, New York, NY, USA, 2017. ACM.

[10] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

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

[12] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. SIGMOD '09, pages 19–30, 2009.

[13] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. SIGMOD '96, pages 294–305.

[14] W. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *PVLDB*, pages 1954–1965, 2013.

[15] W. Qardaji, W. Yang, and N. Li. Priview: Practical differentially private release of marginal contingency tables. SIGMOD '14, pages 1435–1446, 2014.

[16] R. G. J. G. Steven Ruggles, Katie Genadek and M. Sobek. Integrated public use microdata series: Version 6.0. http://doi.org/10.18128/D010.V6.0, 2015.

[17] X. Xiao, G. Wang, and J. Gehrke. Differential privacy via wavelet transforms. *IEEE Trans. on Knowl. and Data Eng.*, pages 1200–1214, 2011.

[18] Y. Xiao, J. Gardner, and L. Xiong. Dpcube: Releasing differentially private data cubes for health information. In *ICDE*, pages 1305–1308. IEEE, 2012.

[19] J. Xu, Z. Zhang, X. Xiao, Y. Yang, G. Yu, and M. Winslett. Differentially private histogram publication. *PVLDB*, pages 797–822, 2013.

[20] X. Zhang, R. Chen, J. Xu, X. Meng, and Y. Xie. Towards accurate histogram publication under differential privacy. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 587–595. SIAM, 2014.