UPA: An Automated, Accurate and Efficient Differentially Private Big-data Mining System

Tsz On Li*, Jianyu Jiang*, Ji Qi*, Chi Chiu So*, Jiacheng Ma*, Xusheng Chen*, Tianxiang Shen*,
Heming Cui*, Yuexuan Wang†*, and Peng Wang‡
*Department of Computer Science, The University of Hong Kong
†College of Computer Science and Technology, Zhejiang University, China
‡Theory Lab, 2012 Labs, Huawei Technologies, Co. Ltd., Hong Kong
Email: {toli2, jyjiang, jqi}@cs.hku.hk, {kelccso, u3533156,chenxus,stx635}@connect.hku.hk,
{heming, amywang}@cs.hku.hk, wang.peng6@huawei.com

***Abstract*—In the era of big-data, individuals and institutions store their sensitive data on clouds, and these data are often analyzed and computed by MapReduce frameworks (e.g., Spark). However, releasing the computation result on these data may leak privacy. Differential Privacy (DP) is a powerful method to preserve the privacy of an individual data record from a computation result. Given an input dataset and a query, DP typically perturbs an output value with noise proportional to *sensitivity*, the greatest change on an output value when a record is added to or removed from the input dataset. Unfortunately, directly computing the sensitivity value for a query and an input dataset is computationally infeasible, because it requires adding or removing every record from the dataset and repeatedly running the same query on the dataset: a dataset of one million input records requires running the same query for more than one million times.**

This paper presents UPA, the first automated, accurate, and efficient sensitivity inferring approach for big-data mining applications. Our key observation is that MapReduce operators often have commutative and associative properties in order to enable parallelism and fault tolerance among computers. Therefore, UPA can greatly reduce the repeated computations at runtime while computing a precise sensitivity value automatically for general big-data queries. We compared UPA with FLEX, the most relevant work that does static analysis on queries to infer sensitivity values. Based on an extensive evaluation on nine diverse Spark queries, UPA supports all the nine evaluated queries, while FLEX supports only five of the nine queries. For the five queries which both UPA and FLEX can support, UPA enforces DP with five orders of magnitude more accurate sensitivity values than FLEX. UPA has reasonable performance overhead compared to native Spark. UPA's source code is available on https://github.com/hku-systems/UPA.

I. Introduction

In the era of big-data, voluminous data records are stored on clouds and analyzed by third-parties for diverse data-mining tasks, such as product recommendations. However, the output values of these analytical tasks often contain user sensitive information and hence require protection [1, 2]. For instance, researchers successfully extracted user sensitive information by analyzing the user data of Facebook and Netflix [3, 4].

Differential Privacy [5–13] (DP) is a powerful theoretical framework to preserve the privacy of individual records in a dataset. Essentially, DP perturbs a query's output values with noise to hide the existence of an individual record. The noise is proportional to ***sensitivity***, the greatest change on an output value when an individual record is added to or removed from an input dataset. This ensures that an attacker cannot guess whether a specific individual record is in the dataset.

Given a query and an input dataset, a sensitivity value must be precisely computed. If it is too large (inaccurate), excessive noise will be added to an output value and enormously degrade the accuracy of the output; if it is too small, the noise will be insufficient to preserve the privacy of all individual records in the dataset (DP fails to be enforced).

Unfortunately, computing a precise sensitivity value is challenging because a sensitivity value varies greatly among different queries and input datasets. Brute-force approach (i.e., repeatedly running the same query with a data record being added to or removed from the same dataset each time) is currently the only method that can compute the exact sensitivity value for arbitrary queries and input datasets, yet it incurs prohibitive performance overhead [14, 15]: if a dataset has one million records, then brute-force approach has to run the same query more than one million times for computing the sensitivity. Since no existing approach can efficiently compute a sensitivity value for diverse queries, existing data-mining systems that enforce DP either support only a limited set of mining operators whose sensitivity can be theoretically analyzed (e.g., `count` whose sensitivity is one for any input dataset) [16, 17], or require the sensitivity values to be manually estimated by experts [18–27], which is tedious and error-prone [18].

To reduce manual effort in inferring sensitivity, a latest approach FLEX [14] does static analysis to estimate a sensitivity value for SQL queries by analyzing the types of SQL operators (e.g., `Join`) in an SQL query and an input dataset's metadata (e.g., number of data records in each input column). However, FLEX is inaccurate because it considers only the composing set of operators in a query, and ignores the actual query logic (e.g., data and control flow of a query). Moreover, FLEX only supports a limited set of SQL operators (i.e., `Select`,`Join`,`Filter` and `Count`), and thus does not support diverse data mining queries. Overall, despite much effort, an automated, accurate and efficient sensitivity estimation approach for big-data queries is still missing.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00064

Our key observation is that big-data operators (e.g., MapReduce operators) are often written in a commutative and associative manner to enable parallelism and fault tolerance among computers. Commutativity means that input data records can be computed by union operations (e.g., `Map`) regardless of their order in a dataset. Associativity means that data can be computed in partitions and then being aggregated (e.g., `Reduce`), as if the data are computed as a whole. Since computing sensitivity of a query requires repeatedly running the same query on mostly overlapped input data records (in the brute-force approach, a query repeatedly runs on the same dataset with only one data record being added or removed from the input dataset each time), Commutativity and Associativity of big-data operators allow reusing intermediate results computed from the overlapped inputs. Moreover, unlike FLEX, we can estimate a sensitivity value accurately from a query's concrete logic. Therefore, the efficiency and accuracy of computing sensitivity can be greatly enhanced.

We propose Union Preserving Aggregation (UPA), the first automated, accurate and efficient system to enforce DP in big-data mining tasks. Given a query and an input dataset, UPA first randomly samples a small fraction of records from an entire input dataset as the differing records (the records that are to be added to or removed from the input dataset). Commutativity allows the differing records (the sampled data records) and overlapped records (the un-sampled data records) to be computed disjointly regardless of their order in the input dataset. Associativity allows the computation results of the overlapped records and the differing records to be aggregated, as if they are computed together without being partitioned. These two properties allow UPA to resue the intermediate result of the computation on the overlapped records, to efficiently and concurrently compute the output value of the query on the entire input dataset, as well as the output values of the query with a differing record being added or removed from the input dataset. UPA then infers a sensitivity value based on these output values.

However, same as existing DP systems [18–21] that manually infer a sensitivity value, UPA faces a challenge to provide DP guarantee. Specifically, the inferred sensitivity may be smaller than the true sensitivity, which implies that the added noise may not cover the change (influence) on an output value for some input data records (i.e., DP is not guaranteed for the entire dataset). To solve this challenge, we propose RANGE ENFORCER, an algorithm to constrain the output range of a query, such that the sensitivity value inferred by UPA is always sufficiently large (i.e., always provides DP guarantee to the entire dataset). Since the same constrained output range has to be applied to the same query (queries that have the same input-output mapping) to enforce DP, RANGE ENFORCER leverages the Commutativity and Associativity properties to efficiently identify the similarity between queries and apply the same constrained output ranges to the same queries. We carry a proof for UPA's DP guarantee in §IV-C.

We implemented UPA and integrated it with Spark [28]. UPA eliminates the modification of Spark queries by developing a set of DP-enabled, Spark-compatible MapReduce operators. We evaluated UPA on nine diverse mining queries, including two Spark user-defined queries (e.g., KMeans [18] and Linear Regression [18]) and seven SparkSQL queries. These queries included all five open-source queries evaluated in FLEX [14]. We compared UPA with vanilla Spark and FLEX [14] in efficiency and accuracy. Evaluation shows that:

- UPA computes the sensitivity values for all evaluated queries automatically (neither modification on a query nor expert knowledge on an input dataset is required).
- UPA supports all the nine big-data queries, while FLEX supports only five queries composed of `Select`,`Join`,`Filter` and `Count` operators in SparkSQL.
- UPA is accurate: UPA's inferred sensitivity values, compared to the sensitivity values computed by brute-force approach, merely had 3.81% Root Mean Square Error (RMSE) on average for all queries. For all the five queries that both UPA and FLEX support, UPA's RMSE was one to five orders of magnitude smaller than FLEX.
- UPA on average had only 77.6% performance overhead compared to vanilla Spark. For larger dataset sizes, UPA's performance overhead even gradually decreased.

Our main contribution is UPA, the first automated, accurate, and efficient DP big-data computation system. UPA supports various big-data mining algorithms without modification of the queries or expert knowledge in manually inferring a sensitivity value. UPA can be applied to Spark as well as other big-data platforms (e.g., Hadoop [29], DryadLINQ [30]), greatly promoting the adoption of DP and improving privacy of user data in real-world big-data analytics.

The rest of this paper is structured as follows. §II introduces the background of Differential Privacy, FLEX and MapReduce. §III gives an overview of the architecture of UPA. §IV and §V describe the design and implementation of UPA. We show UPA's evaluation results in §VI. We introduce UPA's related work in §VII. We discuss and conclude in §VIII.

II. BACKGROUND

A. Differential Privacy

Differential privacy [5] (DP) is a privacy preserving technique that adds randomness (typically random noise) to an output value of a query on an input dataset, in order to cover up the potential change on the output value due to a data record being added to or removed from the input dataset (any pair of datasets that differ by only one record is called **neighbouring dataset**). Since such change on the output value reveals the presence of the data record and leaks privacy, to cover up such change for all data records in the input dataset, DP adds noise to an output value of a query proportional to ***sensitivity***.

There are three different types of sensitivity in DP context, namely global sensitivity, local sensitivity, and smooth sensitivity. **Global Sensitivity** [5] refers to the greatest difference in the output values of a query on any pair of neighbouring datasets from all possible input datasets of the query [5]. However, global sensitivity often adds too much noise to an output value because it provides protection to data records that do not even exist in the actual input dataset.

In contrast to global sensitivity, **local sensitivity** [5] is the greatest difference on an output value of a query on any pair of neighbouring datasets exists in the actual input dataset [5]. Hence, local sensitivity provides higher utility [5] than global sensitivity. Calibrating noise based on local sensitivity is sufficient to provide DP guarantee to each individual record in an input dataset, which satisfies *Individual Differential Privacy* (iDP) [31]. UPA infers local sensitivity in order to enforce iDP. The definition of local sensitivity and iDP is as follows:

Definition II.1. Local Sensitivity: given an input dataset $x \in D$ and a query $f : D \to R$, for any (neighbouring) dataset $y \in D$ that differs x by one data record (i.e., $d(x, y) = 1$), the local sensitivity is

$$LS_f(x) = \max_{y:d(x,y)=1} |f(x) - f(y)|$$

Definition II.2. Individual Differential Privacy (iDP): given an input dataset x and a query f, a mechanism $\kappa(\cdot)$ that randomizes the output value of f satisfies ϵ-individual differential privacy if, for any x's neighbouring dataset x' and any output value o returned by $\kappa(\cdot)$:

$$exp(-\epsilon) \leq \frac{P(\kappa(x) = o)}{P(\kappa(x') = o)} \leq exp(\epsilon)$$

iDP offers much higher utility than DP while providing the same privacy guarantee as standard DP [5] for any individual's data record, which satisfies the goal of UPA and other DP systems [32–34]. This paper does not aim to provide DP guarantee to a group of individuals, although UPA can be extended to do so in future work (§VI-E).

B. FLEX

Smooth sensitivity [35] is proposed to provide DP guarantee to both individuals and a group of data records. It calibrates noise based on the maximum local sensitivity when any number of records is added to or removed from an input dataset. FLEX [14] automatically infers the local sensitivity and smooth sensitivity of `count` queries in SQL that have `Join` operators. Although FLEX also describes how it can be extended to queries with other SQL operators including `SUM`, `AVG`, `MIN` and `MAX`, these extensions are informally discussed as the "possible extensions" in FLEX's paper [14], and FLEX's paper does not describe how are they realised. Thus, this paper only compares to FLEX on queries with `count` operators.

For a query that counts the number of joined records after joining two columns of a dataset, FLEX infers the sensitivity by multiplying the frequencies of the most frequently-occurring item from each of the two columns, because removing a record from the dataset can at most affect such a number of joined records. That is, FLEX assumes worst case happens and infers the worst case sensitivity, which is the upper bound of the true sensitivity.

Nonetheless, FLEX has three limitations. First, it only supports `count` queries because its static analysis approach can only be applied to frequency computation (i.e., `count`); FLEX does not support non-count queries such as arithmetic and machine learning queries. Second, FLEX only works on SQL because it relies on relational algebra to compute the sensitivity when a query has multiple `Join` operators.

Third, since FLEX considers only the composition sets of SQL operators and ignores the actual query logic of these operators (i.e., data and control flow in a query), FLEX is inaccurate in two aspects. First, FLEX does not consider the actual join keys of records in a dataset. If the most frequently-occurring item from each of the two input columns do not share the same join key, the items of these two columns will not join, causing FLEX overestimates the sensitivity. Second, FLEX does not consider the effect of join condition (i.e., `Filter`) when inferring the worst case sensitivity, while `Filter` often avoids the worst case sensitivity by filtering out some, if not all, the most frequently-occurring items. FLEX's precision is even worse when a query has multiple `Join` and `Filter` (e,g., TPCH16 and TPCH21 in the evaluation of this paper and FLEX's paper [14]), because FLEX multiplies the worst case sensitivity of all `Join`, which causes error magnifies in each `Join` when the worst case does not occur in some `Join`.

To the best of our knowledge, FLEX is the only system that automatically infers sensitivity values. Since the goal of this paper is to provide DP guarantee to individual data records, to make an apple-to-apple comparison, we compared the accuracy of the local sensitivity inferred by FLEX with the local sensitivity inferred by UPA in §VI-B.

C. MapReduce

MapReduce [28, 36] was proposed for massive-data processing tasks such as machine learning. They are widely deployed in e-business, finance, medical analysis, and many other fields. These MapReduce-related platforms (e.g., Spark[28]) usually adopt functional operators such as `Map` and `Reduce`. To enable parallelism and fault-tolerance among Spark computing nodes, these operators are often written in a commutative and associative manner [37]:

Commutativity An operation is commutative if the order of input data records does not affect the output value [37]. Both Map and Reduce are commutative. Commutativity is important for parallelization because it allows the parallel scheduler to assign map or reduce tasks to computing nodes in any order.

Associativity Suppose D_1 and D_2 are two disjoint datasets and $D = D_1 \cup D_2$, an operation O is associative iff $O(O(D_1) \cup O(D_2)) = O(D_1 \cup D_2)$ [37]. Map and Reduce are associative because associativity allows input data to be parallelly processed on individual computers before they are aggregated.

Airavat [19] is a related DP system which avoids data leakage via OS channels (e.g., network/IO channel) by modifying a JVM to restrict the expressiveness of a MapReduce query. However, to enforce end-to-end DP, Airavat still requires a sensitivity value to be manually inferred (i.e., Airavat is not automatic in enforcing end-to-end DP). Since UPA and

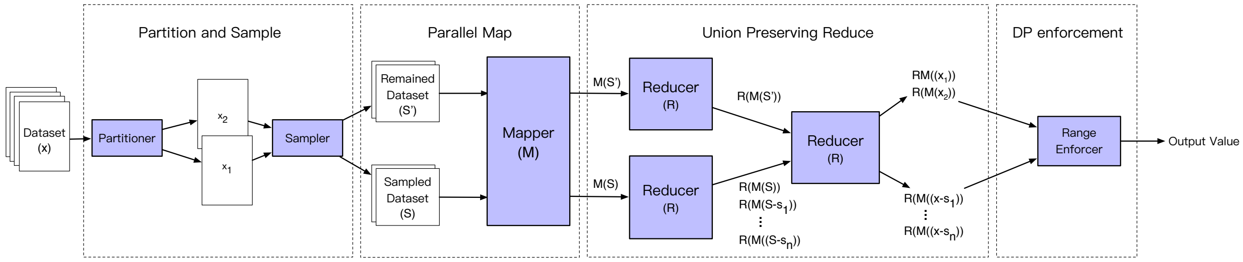

Fig. 1: Architecture of UPA. Union Preserving Aggregation components are shaded and in purple. M and R represent the map and reduce function respectively. x_i is the ith partition in x, and s_i is the ith data record in S.

Airavat are orthogonal (automatic vs manual) in inferring sensitivity, this paper does not compare UPA with Airavat. To the best of our knowledge, no existing DP system can automatically infer sensitivity values for MapReduce queries, so this paper aims to build the first one.

Although UPA is implemented on Spark, UPA can also be extended to other big-data processing systems (e.g., Camdoop [38], DryadLINQ [30], Pregel [39] and Angel [40]), because these systems also have commutative and associative operators in order to enable parallelism and fault-tolerant for big-data processing.

III. Overview

In UPA's threat model, data providers and UPA are trusted, and datasets are maintained in UPA. Data analysts are adversaries who know the specific value of some attributes of an individual's data record, and keep conducting queries on one dataset provided by the data providers. Specifically, the analyst uses this specific value to filter out an individual's record from the dataset and submits the same query repeatedly (any two queries are considered the same if they have the same input-output mapping). Then, the data analyst may infer the existence of an individual's record in the dataset from the queries' outputs. UPA defends against such attacks by enforcing iDP.

Figure 1 shows UPA's architecture and workflow. A data analyst submits a big-data query to UPA for computing a dataset, and UPA automatically and concurrently infers a local sensitivity value and computes the query's output value. UPA then returns a noisy output value to the data analyst, with noise calibrated based on the local sensitivity value for enforcing iDP.

We take Linear Regression (LR) as an example to illustrate UPA's architecture and workflow. LR is a typical big-data mining query which uses Stochastic Gradient Descent (SGD) to update the parameters of a linear model. LR computes an SGD value for each data record in the input dataset (done in Mapper M). Then, LR sums up all the SGD's value, and the sum is used to update the linear model's parameters (done in Reducer R). The output of LR is the updated linear model. Our evaluation (§VI) shows that LR's output is different for neighbouring input datasets (§II-A). This indicates that enforcing iDP is crucial to preserve data privacy for LR.

According to Figure 1, UPA's workflow can be divided into four phases: *Partition and Sample*, *Parallel Map*, *Union Preserving Reduce* and *iDP Enforcement*. The first phase is *Partition and Sample*, which partitions and samples the data records of the input dataset. In this phase, UPA has a parameter n: n is for inferring an accurate sensitivity local value. After an LR query is submitted to UPA by a data analyst, UPA's partitioner first partitions the input dataset x into two partitions (x_1 and x_2). UPA later computes the output values for each of these two partitions $(R(M(x_1))$ and $R(M(x_2)))$, and they are analyzed by UPA's Range Enforcer for enforcing iDP before UPA returns the LR's output to the data analyst. UPA then uniformly samples totally n data records from all records in all these partitions (n is 1000 by default in UPA). This default value is statistically sufficient to infer an accurate sensitivity value [41, 42] (see §IV-A). We denote the sampled data records as S and the remaining (unsampled) data records as S'. The values of n are evaluated in §VI-C and §VI-D, and the default values are robust to general queries.

The second phase is *Parallel Map*. UPA applies the Mapper M to both the sampled data records (S) and the remaining data records (S'), in order to compute an SGD value for each record in S and S'.

The third phase is *Union Preserving Reduce*, which applies the Reducer R on $M(S')$ and $M(S)$ to compute the LR's output on the entire input dataset (i.e., $R(M(x))$), LR's output on x's partitions $(R(M(x_1)),R(M(x_2)))$, and LR's output on the entire input dataset with each of the sampled records being excluded $(R(M(x-s_1)),...,R(M(x-s_n)))$. To ease discussion, $x-s_1,...,x-s_n$ is referred as the **sampled neighbouring datasets** of x.

UPA first computes $R(M(S'))$, because it can be reused to compute all these output values by leveraging MapReduce's Commutativity and Associativity. Then, UPA computes the output values on the sampled neighbouring datasets of x $(R(M(x-s_1)),...,R(M(x-s_n)))$. UPA does so by iteratively removing each record in S and applies the Reducer R to the rest of the records in S (i.e., UPA computes $R(M(S-s_i)), 1 \leq i \leq n$); then, UPA applies the Reducer R to both $R(M(S-s_i))$ and $R(M(S'))$, in order to compute $R(M(x-s_i))$. $R(M(x-s_1)),...,R(M(x-s_n))$ is therefore efficiently computed and are used to infer a sensitivity value for x. In short, UPA greatly improves the efficiency of inferring a sensitivity value by reusing $R(M(S'))$. Based on the values of $R(M(S'))$ and all $R(M(S{-}s_i))$, LR's output on the entire input dataset and on x's partitions are also obtained. Details will be discussed in §IV.

The last phase is *iDP Enforcement*. UPA infers a local sensitivity value based on $R(M(x-s_1))$,...,$R(M(x-s_n))$, according to the definition of local sensitivity (Definition II.1). Then, to enforce iDP with the inferred local sensitivity, UPA needs to identify whether LR has been submitted before, and if so, whether the current input dataset and the previous input datasets are neighbouring (Details are in §IV-B).

To do so, UPA's Range Enforcer first compares the output values of LR on x's partitions ($R(M(x_1))$ and $R(M(x_2))$), with the output values of any prior query q on its input dataset's partitions (let x' denote q's input dataset). Intuitively, if all the output values of LR on x's partitions are different from the output values of q on x''s partitions (i.e., $LR(x_1) \neq q(x_1')$ and $LR(x_2) \neq q(x_2')$), LR is not the same as q, or x and x' differ by at least two records, which is trivial to be handled in UPA. The malicious case is that at least one partition has the same output value (i.e., $LR(x_1) = q(x_1')$) or $LR(x_2) = q(x_2')$), which means LR may be the same as q, and x and x' may differ by just one record. UPA avoids this malicious case by removing at least two records from x (let x'' denote x with records removed by UPA), and return $LR(x'')$ to the data analyst after adding noise (Algorithm 2 in §IV-B). We carry a proof of UPA's iDP guarantee in §IV-C.

IV. UPA's Theories and Tools

This section detailedly explains how UPA infers a local sensitivity value and how UPA's Range Enforcer enforces iDP. Then it proves UPA enforces (ϵ)-iDP.

A. Inferring Local Sensitivity

Given a query f (composed of a Mapper M and a Reducer R) and an input dataset $x \in D$, where D is the possible value range of any data record in the input dataset x, UPA infers a local sensitivity value of $f(x)$ (i.e., $LS_f(x)$ in Definition II.1). However, computing $LS_f(x)$ in a brute-force manner incurs prohibitive performance overhead (§I). To make inferring local sensitivity and enforcing iDP feasible and efficient, UPA samples the output values of f on x's neighbouring dataset to infer $LS_f(x)$, and constrains the output range of f on x and its neighbouring dataset, such that the inferred local sensitivity is an upperbound of the ground-truth local sensitivity of $f(x)$ (a prerequisite for proving UPA's iDP guarantee, see §IV-C).

Algorithm 1 shows how UPA efficiently samples the output values of f on x's neighbouring dataset, including the four phases in §III. The first phase is *Partition and Sample* (line 1∼2). UPA splits x into two partitions ($\{x_i\}_{i=1}^2$). UPA later computes an output value of f on each of these two partitions for UPA's Range Enforcer to enforce iDP (Section IV-B). Then, UPA uniformly samples n data records (with sample size n as 1000) from all these two partitions (let $\{s_i^{(x)}\}_{i=1}^n$ denote the n sampled records from x), as well as n data records from D but not in x (let $\{s_i^{(\overline{x})}\}_{i=1}^n$ denote the n sampled records from D but not in x).

The sample size n is set as 1000 by default because local sensitivity is often theoretically regarded as a random variable that follows normal distribution [15, 34, 43] (i.e., most data records of an input dataset have small influence on the output value, only few outliers exist). Hence, to precisely identify the local sensitivity (i.e., a normal distribution) of a query and an input dataset, the 1000 sample size has been proven statistically sufficient [41, 42]. Our evaluation also shows that the 99th percentile of the normal distribution inferred by this sample size covers most (more than 98.9% for eight out of nine evaluated queries with input datasets of twenty thousand data records) output values of all neighbouring datasets (Figure 3).

For an input dataset less than 1000 records, n has to be set lower than 1000. In this case, n should be set as the total number of data records in the dataset, so the output values of f on all of x's neighbouring datasets are obtained (i.e., the exact local sensitivity value of $f(x)$ is obtained). Nevertheless, since UPA aims to provide iDP for big-data processing tasks (whose input datasets often consist of more than millions data records [28, 36, 38]), we assume input datasets always have more than 1000 data records throughout this paper (i.e., n is always set as 1000).

The second phrase is *Parallel Map* (line 4∼6), which applies the Mapper M to the sampled data records $\{s_i^{(x)}\}_{i=1}^n$, $\{s_i^{(\overline{x})}\}_{i=1}^n$ and the remaining records $\{s_i'\}_{i=1}^{|x|-n}$ (where s_i' is a data record in the remaining dataset S', defined in §III). Given these three sets of records, this phase passes the M's output on these sets ($\{M_i^{(s_x)}\}_{i=1}^n$, $\{M_i^{(s_{\overline{x}})}\}_{i=1}^n$ and $\{M_i^{(s')}\}_{i=1}^{|x|-n}$) to the next phase.

The third phase is *Union Preserving Reduce* (line 7∼16). UPA first computes the value of R on $\{M_i^{(s')}\}_{i=1}^{|x|-n}$ (line 7∼8), because this value can be reused to compute f's output value on x, f's output value on x_1 and x_2, and f's output value on the "sampled neighbouring datasets" of x (i.e., $x - s_1$,..., $x - s_n$, define in §III). Then, UPA computes the output values of f on these sampled neighbouring datasets. These output values are $\{o_i\}_{i=1}^n$ and $\{\overline{o}_i\}_{i=1}^n$, where o_i is f's output value on x without the ith record in $\{s_i^{(x)}\}_{i=1}^n$ (line 10∼11), $\overline{o}_i$ is f's output value on x with the ith record in $\{s_i^{(\overline{x})}\}_{i=1}^n$ (line 13∼14). Finally, UPA computes the output value of f on the two partitions of x ($\{x_i\}_{i=1}^2$) (line 15∼16).

The last phase is *iDP Enforcement* (line 17∼22). UPA infers a local sensitivity value based on $\{o_i\}_{i=1}^n$ and $\{\overline{o}_i\}_{i=1}^n$. Specifically, UPA uses Maximum Likelihood Estimation (MLE) to identify the underlying normal distribution of $\{o_i\}_{i=1}^n$ and $\{\overline{o}_i\}_{i=1}^n$, and computes the difference between the 1th and 99th percentile of the normal distribution as the local sensitivity value of $f(x)$ (the 1th and 99th percentile of a normal distribution are often regarded as extreme values of the normal distribution [44, 45]). UPA's Range Enforcer also uses the 1th and 99th percentile of the normal distribution to constrain the output range of f on x and x's neighbouring dataset in order to enforce iDP. Detailed discussion is in Section IV-B.

```
Algorithm 1: Inferring Sensitivity
Input: $M$: Mapper, $R$: Reducer, $x$: input dataset, $n$: sample
       size,
$\{x_i\}_{i=1}^{2} \leftarrow Partition(x)$
$\{s_i^{(x)}\}_{i=1}^{n}, \{s_i^{(\overline{x})}\}_{i=1}^{n} \leftarrow Sample(\{x_i\}_{i=1}^{2}, n)$
$\{s_i'\}_{i=1}^{|x|-n} \leftarrow \{x_i\}_{i=1}^{2}/\{s_i^{(x)}\}_{i=1}^{n}$
$\{M_i^{(s_x)}\}_{i=1}^{n} \leftarrow M(\{s_i^{(x)}\}_{i=1}^{n})$
$\{M_i^{(s_{\overline{x}})}\}_{i=1}^{n} \leftarrow M(\{s_i^{(\overline{x})}\}_{i=1}^{n})$
$\{M_i^{(s')}\}_{i=1}^{|x|-n} \leftarrow M(\{s_i'\}_{i=1}^{|x|-n})$
$\{R_i^{(s')}\}_{i=1}^{2} \leftarrow ReduceByPar(R, \{M_i^{(s')}\}_{i=1}^{|x|-n})$
$R^{(s')} \leftarrow R(\{R_i^{(s')}\}_{i=1}^{2})$
Initialise an empty array $\{o_i\}_{i=1}^{n}$
for $p \leftarrow 1$ to $n$ do
    $o_p \leftarrow R(R^{(s')}, R(\{M_i^{(s_x)}\}_{i=1}^{n}/M_p^{(s_x)}))$
Initialise an empty array $\{\overline{o}_i\}_{i=1}^{n}$
for $p \leftarrow 1$ to $n$ do
    $\overline{o}_{p,q} \leftarrow R(Output, M_p^{(s_{\overline{x}})})$
$\{R_i^{(s)}\}_{i=1}^{2} \leftarrow ReduceByPar(R, \{M_i^{(s)}\}_{i=1}^{n})$
$\{R_i^{(x)}\}_{i=1}^{2} \leftarrow ReduceByPar(R, \{\{R_i^{(s')}\}_{i=1}^{2}, \{R_i^{(s)}\}_{i=1}^{2}\})$
sampNeigh $\leftarrow \{\{o_i\}_{i=1}^{n}, \{\overline{o}_i\}_{i=1}^{n})\}$
normalSamp $\leftarrow$ MLE(sampNeigh)
$(lowerPercentile, upperPercentile) \leftarrow$
  percentile(normalSamp,1,99)
localSen $\leftarrow$ upperPercentile - lowerPercentile
outRange $\leftarrow$ (lowerPercentile,upperPercentile)
Output $\leftarrow$ RANGE ENFORCER
  $(R,\{M_i^{(s_x)}\}_{i=1}^{n},R^{(s')},\{R_i^{(x)}\}_{i=1}^{2},outRange)$
Output: Output + Lap(localSen)
```

B. *Enforcing iDP with* RANGE ENFORCER

We first give a high-level overview of RANGE ENFORCER, an algorithm for UPA to enforce iDP with the inferred local sensitivity value. As introduced in §III, RANGE ENFORCER's objective is to detect whether a query submitted to UPA has also been submitted to UPA before, and if so, whether the data analyst has maliciously added or removed one record (i.e., whether the analyst is conducting an attack).

RANGE ENFORCER does so by identifying whether the output values of two queries on their input dataset's partitions are different. If the output values are all different, the two queries' input dataset must differ by at least two records (i.e., the analyst is not conducting an attack). Otherwise, the two queries' input dataset may differ by just one record (i.e., the analyst is likely conducting an attack). In this case, RANGE ENFORCER removes at least two records from a query's input dataset, so that the two queries' input dataset differ by at least two records (i.e., RANGE ENFORCER avoids the data analyst conducting an attack). Hence. RANGE ENFORCER provides iDP guarantee to the input dataset (Proof in §IV-C).

In detail, to enforce iDP, UPA needs to make sure the inferred local sensitivity value is an upper bound of the actual local sensitivity of f on x ($LS_f(x)$). Although Algorithm 1 efficiently infers a local sensitivity value based on the output values of f on the sampled neighbouring datasets of x ($\{o_i\}_{i=1}^{n}$ and $\{\overline{o}_i\}_{i=1}^{n}$), there is no guarantee that the inferred local sensitivity value is an upper bound of $LS_f(x)$ (i.e., there is no guarantee that UPA can provide iDP guarantee to all data records in x with the inferred local sensitivity value).

```
Algorithm 2: RANGE ENFORCER
Input: $R$: Reducer, $\{M_i^{(s_x)}\}_{i=1}^{n}$: Mapped value of the
       sampled data records, $R^{(s')}$: Reduced value of the
       data records which are not sampled $\{R_i^{(x)}\}_{i=1}^{2}$:
       Reduced value on $x$'s partitions, outRange: the output
       range inferred from the output values of sampled
       neighbouring datasets, $sampNeigh$: the output
       values of $x$'s sampled neighbouring datasets
Variables: $p$: number of queries submitted before,
           $\{R_{i,j}^{(x')}\}_{i,j=1}^{p,2}$: the reduced value of $q_i$ on dataset
           $x'$'s $j$th partition ($x' \in D$)
q $\leftarrow$ 0
if $p > 0$ then
    for $i \leftarrow 1$ to $p$ do
        diffNum $\leftarrow$ 0
        for $j \leftarrow 1$ to 2 do
            if $R_{i,j}^{(x')} \neq R_j^{(x)}$ then
                diffNum $\leftarrow$ diffNum +1
        while diffNum $< 2$ do
            diffNum $\leftarrow$ 0
            $M' \leftarrow$ remove two data records $\{M_i^{(s)}\}_{i=1}^{n}$
            $\{R_i^{(s)}\}_{i=1}^{2} \leftarrow ReduceByPar(R, M')$
            $\{R_i^{(x)}\}_{i=1}^{2} \leftarrow$
              $ReduceByPar(R, \{\{R_i^{(s')}\}_{i=1}^{2}, \{R_i^{(s)}\}_{i=1}^{2}\})$
            for $j \leftarrow 1$ to 2 do
                if $R_{i,j}^{(x')} \neq R_j^{(x)}$ then
                    diffNum $\leftarrow$ diffNum +1
Output $\leftarrow R(\{R_i^{(x)}\}_{i=1}^{2})$
if Output $>$ outRange.max or Output $<$ outRange.min then
    Output $\leftarrow$
      randomBetween(min(outRange), max(outRange))
for $i = 1$ to 2 do
    $R_{p+1,i}^{(x')} \leftarrow R_i^{(x)}$
$p = p + 1$
Output: Output
```

This problem is also encountered by existing DP systems [18–21] which require data analysts to manually infer a *global* sensitivity value (§II-A). It is because there is also no guarantee that the manually inferred global sensitivity is an upper bound of the actual global sensitivity value (i.e., there is no guarantee that these systems can provide DP guarantee to all records in an input dataset with the inferred global sensitivity).

To address this problem, these DP systems require data analysts to manually infer an output range of f (denote this output range as $\hat{O}_f$). $\hat{O}_f$ is a tuple of two values: the maximum output value $max(\hat{O}_f)$ and the minimum output value $min(\hat{O}_f)$. Then, all output values of f will be constrained within $\hat{O}_f$. By doing so, the output value of f on any neighbouring datasets must be within $\hat{O}_f$, which means the global sensitivity of f is at most $max(\hat{O}_f) - min(\hat{O}_f)$. Hence, an upper bound of f's global sensitivity value is obtained as $max(\hat{O}_f) - min(\hat{O}_f)$. With this upper bound, these DP systems can provide DP guarantee to all records in an input dataset.

RANGE ENFORCER is inspired by this method to provide iDP to an input dataset with the inferred local sensitivity.

RANGE ENFORCER constrains the output range of f on x and x's neighbouring datasets within $\hat{O}_f$: $\hat{O}_f$ consists of the minimum and maximum output value of f on the sampled neighbouring datasets of x (i.e., $\{o_i\}_{i=1}^n$ and $\{\overline{o}_i\}_{i=1}^n$), so the local sensitivity is inferred as the greatest difference between $\{o_i\}_{i=1}^n$ and $\{\overline{o}_i\}_{i=1}^n$. Whenever a query is submitted to UPA, RANGE ENFORCER detects if the query is f and if the input dataset is x or x's neighbouring datasets. If so, RANGE ENFORCER constraints the output value of the query on the input dataset within $\hat{O}_f$, same as existing DP systems. Otherwise, RANGE ENFORCER infers a new output constraint for the query and the input dataset.

However, automatically and precisely identifying whether two queries are the same and whether their input dataset is neighbouring is challenging (i.e., identifying an attack is challenging). It is because an attacker can submit queries that have different syntax to UPA to conduct an attack. To address this challenge, our observation is that big-data operators often process each data record in a dataset independently (i.e., there is no shared state between the processing of each data record), because big-data operators are commutative and associative. Hence, if two queries are the same, and their input dataset only differs by a few data records (i.e., their input dataset is mostly overlapped), the output value of these two queries on the overlapped partition of their input dataset is also the same. Based on this observation, we propose RANGE ENFORCER, a technique to identify and avoid an attack based on the output values of two queries on their input dataset's partitions.

Specifically, RANGE ENFORCER divides data provider's dataset D into two partitions (i.e., D_1 and D_2). When a data analyst submits a query f for querying $x \in D$, RANGE ENFORCER computes the output value of f on the two partitions of x (i.e., $f(x_1)$ and $f(x_2)$, where $x_1 \in D_1$ and $x_2 \in D_2$). Then, RANGE ENFORCER compares these output values with the output values of any previous query f' submitted to UPA on the two partitions of their input dataset $x' \in D$ ($f'(x'_1)$ and $f'(x'_2)$, where $x'_1 \in D_1$ and $x'_2 \in D_2$). The comparison (Algorithm 2) includes the following two complete cases:

Case 1: $f(x_1) \neq f'(x'_1)$ and $f(x_2) \neq f'(x'_2)$, which means x' and x must differ by at least two data records. Since x and x' differ by more than one data record, x and x' are not neighbouring (i.e., $f'(x')$ and $f(x)$ is not an attack). Hence, RANGE ENFORCER infers and enforces a new output constraint for $f(x)$, and returns $f(x)$ to the data analyst after adding noise.

Case 2: $f(x_1) = f'(x'_1)$ or $f(x_2) = f'(x'_2)$, which means $f(x)$ and $f'(x')$ could be an attack. To avoid this attack, RANGE ENFORCER iteratively removes two records from x (let x'' denote x with records removed by RANGE ENFORCER), such that $f(x''_1) \neq f'(x'_1)$ and $f(x''_2) \neq f'(x'_2)$. Since both partitions of $f(x'')$ and $f'(x')$ have different output values (i.e., $f(x'')$ and $f'(x')$ is not an attack), RANGE ENFORCER infers and enforces a new output constraint for $f(x'')$, and returns $f(x'')$ to the data analyst after adding noise.

Algorithm 2 shows the workflow of RANGE ENFORCER. RANGE ENFORCER first compares the two partitions' output value of the current query ($\{R_j^{(x)}\}_{j=1}^2$) with the two partitions' output value of the ith query submitted to UPA before ($\{R_{i,j}^{(x')}\}_{j=1}^2$) (line 3∼15). Then, it counts the number of different values (diffNum) between $\{R_j^{(x)}\}_{j=1}^2$ and $\{R_{i,j}^{(x')}\}_{j=1}^2$ (line 5∼7). Finally, based on diffNum, RANGE ENFORCER determines if the current query is the same as the ith query, and if the current input dataset and the ith query's input dataset are neighbouring (i.e., diffNum < 2). If so, RANGE ENFORCER iteratively removes two data records from $\{M_i^{(s_x)}\}_{i=1}^n$, and recomputes the output values of the current query on the two partitions of x ($\{R_i^{(x)}\}_{i=1}^2$) , in order to force the current input dataset and the ith query's input dataset to be non-neighbouring (line 8∼15). Finally, Algorithm 2 computes the final output of the current query (line 16), and constrains the final output value within the inferred output range (line 17∼18).

C. *Proof of* UPA*'s iDP guarantee*

We now prove that UPA satisfies ϵ-iDP by proving that UPA infers an upper bound of $f(x)$'s local sensitivity. We first briefly recap the workflow of UPA. UPA samples the output values of f on x's neighbouring datasets. (line 10∼14 of Algorithm 1). Then, UPA infers an output range $\hat{O}_{f(x)}$ based on the output values of UPA'sampled neighbouring datasets, and uses RANGE ENFORCER to constrain the output value of f on x and its neighbouring datasets as $\hat{O}_{f(x)}$ (line 17∼18 of Algorithm 2).

Based on this recap, we first prove the local sensitivity value inferred by UPA is an upper bound of $f(x)$'s local sensitivity (let $LS_f(x)$ denote $f(x)$'s the local sensitivity). This is easy to prove because RANGE ENFORCER constrains the output value of f on x and x's neighbouring datasets within $min(\hat{O}_{f(x)})$ to $max(\hat{O}_{f(x)})$ By doing so, in UPA, the greatest difference between the output value of f on x and f on any neighbouring dataset of x is at most $max(\hat{O}_{f(x)}) - min(\hat{O}_{f(x)})$. Thus, by Definition II.1,

$$\begin{aligned} LS_f(x) &= \max_{y:d(x,y)=1} |f(x) - f(y)| \\ &\leq max(\hat{O}_{f(x)}) - min(\hat{O}_{f(x)})) \end{aligned}$$

Then, we prove that UPA enforces ϵ-iDP with the inferred local sensitivity value. Given a privacy budget ϵ, for any x's neighbouring dataset x' (differs by only one record with x), and an output value o from $\hat{O}_{f(x)}$, according to Definition II.2:

$$\begin{aligned} \frac{P(\text{UPA}_f(x) = o)}{P(\text{UPA}_f(x') = o)} &= \frac{exp(\frac{\epsilon*(o-f(x))}{max(\hat{O}_{f(x)})-min(\hat{O}_{f(x)})})}{exp(\frac{\epsilon*(o-f(x'))}{max(\hat{O}_{f(x)})-min(\hat{O}_{f(x)})})} \\ &\leq exp(\frac{\epsilon * (|f(x) - f(x')|)}{max(\hat{O}_{f(x)}) - min(\hat{O}_{f(x)})}) \\ &\leq exp(\epsilon * \frac{LS_f(x)}{max(\hat{O}_{f(x)}) - min(\hat{O}_{f(x)})}) \\ &\leq exp(\epsilon) \end{aligned}$$

Functions	Description
DP object constructors	
`dpread[T](RDD[T])`	Partition dataset into S and S'.
`dpobject[T](RDD[T], RDD[T])`	Perform RDD operations on S and S'.

`dpobjectKV[K,V]`
`(RDD[(K,V)], RDD[(K,V)])`
	Perform `PairRDDFunctions` operations
	on Key-value S and S'.

dpobject's member functions	
`mapDP(T=>U): dpobject[U]`	Map S and S' and pass them into dpobject.

`reduceDP((T, T) =>T)`
`:(RDD[T],T)`
	Reduce S and S'.
	Return the output value of sampled
	neigbouring datasets and query result.

dpobjectKV's member functions	

`mapDPKV(T =>(K,V))`
`:dpobjectKV[K,V]`
	Map S and S' and pass them
	into dpobjectKV

`reduceByKeyDP((V,V) => V)`
`:dpobject[K,V]`
	Perform ReduceByKey on S and S'.
	Return the output value of sampled
	neigbouring datasets and query result.

`joinDP(dpobjectKV[K,W])`
`:dpobject[K,(V,W)]`
	Perform Join on S and S'.
	Return the output value of sampled
	neigbouring datasets and query result.

TABLE I: API of UPA

Hence, $\frac{P(\text{UPA}_f(x)=o)}{P(\text{UPA}_f(x')=o)} \leq e^{\epsilon}$. Similarly, $\frac{P(\text{UPA}_f(x)=o)}{P(\text{UPA}_f(x')=o)} \geq e^{-\epsilon}$. Therefore, UPA $_f(x)$ satisfies ϵ-iDP. □

V. Implementation

To automatically run Spark queries in UPA without modifying them, UPA provides a complete set of DP-enabled, Spark-compatible operators. The first operator is `dpread`, which partitions and samples the input data loaded into Spark. It returns both the sampled data S and remaining data S' (§III), which are passed to `dpobject` as inputs. `dpobject` is a generic class that carries the map or reduce result of S and S'. In UPA, `mapDP` and `reduceDP` are member functions of `dpobject`, which are emulated to RDD class. Another derivative is `dpobjectKV`, which stores Key-value S and S'. It enables Key-value operations such as `reduceBeyKeyDP` and `joinDP`, which is emulated to `PairRDDFunctions` class.

A. *reduceDP*

`reduceDP` operator first applies the reduce function f to S', let $R_{S'}$ denote the corresponding output value. Then, for each s in both S and the input dataset, `reduceDP` applies f to $R_{S'}$ and $S - s$. This is the computation result of f on x except for the s record (i.e., $f(x - s)$). `ReduceDP` then computes $f(x)$ by applying f on $f(x - s)$ and s. Similarly, for each s in S but not in the input dataset, `ReduceDP` applies f on $f(x)$ and s.

B. *reduceBeyKeyDP*

Similar to `reduceDP`, `reduceBeyKeyDP` first reduces S' with function f. The reduced value and the sampled set S are then separately transformed into a Map and then broadcasted, let $B(R_{S'})$ and $B(S)$ denote the corresponding broadcasted values. The reason to transform them into Map is that it allows UPA's looking up process to be more efficient. Then, for each $s \in S$, f is applied to $B(S) - s$ and $B(R_{S'})$, which is the computation result of f on the D except s.

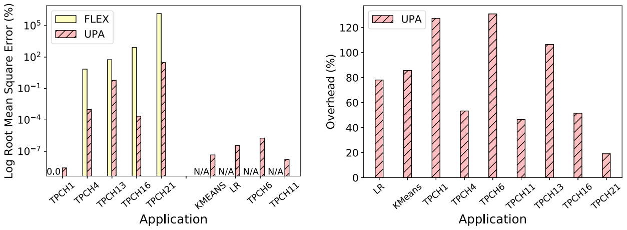

Fig. 2: UPA's and FLEX's RMSE between computed sensitivities (log-scale) and the ground truth (a), and UPA's execution time normalized to vanilla Spark (b). The last four queries are supported by UPA only.

C. *joinDP*

Unlike `reduceDP` and `reduceBeyKeyDP`, `joinDP` takes two input datasets, let D_1 and D_2 denote the two input datasets. Also, let S_1 and S_2 denote the sampled sets of D_1 and D_2 respectively; and let S_1' and S_2' denote the remaining sets of D_1 and D_2 respectively. For each `Join` operator, UPA does two rounds of join and shuffle operations. First, UPA computes the joined tuples of the remaining datasets (i.e., joining S_1' and S_2'). Second, UPA computes the joined tuples of differing tuples (i.e., joins S_1 with S_2', S_1' and S_2, and S_1 with S_2). Therefore, unlike vanilla SparkSQL which does only one shuffle for `Join`, UPA triggers `Join` two times and results in shuffling (exchanging tuples between computers) twice.

Removing one tuple in S_1 or S_2 can result in more than one joined tuples to be removed in the outputs. (`Join` can be either an one-to-one, or one-to-many function). To record the actual influence of removing each tuple in S_1 and S_2 on the number of joined records, the tuples in S_1 and S_2 are given indices in UPA, so that UPA tracks the influence of a differing tuple by finding which joined tuples have the same index as the differing tuple. Overall, by doing so, UPA's accuracy on computing sensitivity values can be much better than FLEX (static analysis), confirmed in our evaluation.

VI. Evaluation

A. *Experiment Setup*

Our evaluation was done on five computers with Intel(R) Xeon(R) CPU E3-1280 v6 with 24 cores, 64GB RAM and 1TB SSD. All computers form a cluster with 40Gbps NIC.

Table II shows our evaluated queries and datasets. We compared UPA with FLEX [14] because UPA and FLEX are the only DP systems that automatically infer a sensitivity value (§II-B). We evaluated UPA on nine diverse mining queries that were evaluated by relevant systems [14, 18, 19], including all the five open-source benchmark queries that were evaluated by FLEX [14]. We used real-world datasets with dataset size of typical big-data applications [28].

To evaluate the accuracy of UPA and FLEX in inferring local sensitivity values (FLEX infers both local sensitivity values and smooth sensitivity values§II-B), we study the Root Mean Square Error (RMSE) between the local sensitivity values computed by UPA or FLEX, and the ground truth local

Query Name	Dataset	Size (GB)	Query Type	Support By UPA	Support By FLEX
TPCHQuery 1	lineitems	114	Count	✓	✓
TPCHQuery 4	lineitems,orders	121	Count	✓	✓
TPCHQuery 13	lineitems,orders	121	Count	✓	✓
TPCHQuery 16	part,supplier,partsupp	117	Count	✓	✓
TPCHQuery 21	supplier,lineitem,orders,nations	133	Count	✓	✓
KMeans	ds1.10 Life Science Data	121	Machine Learning	✓	x
Linear Regression	ds1.10 Life Science Data	121	Machine Learning	✓	x
TPCHQuery 6	lineitems	114	Arithmetic	✓	x
TPCHQuery 11	suppliers,nation,partsupp	115	Arithmetic	✓	x

TABLE II: Evaluation applications and datasets. All datasets are comparable with FLEX.

sensitivity values computed by the brute-force approach (§I) according to Definition II.1.

Same as the evaluation setup of FLEX [14], for each query, we set ϵ as 0.1. Unless specified, we by default sampled 1000 records (n, defined in §III) as the sampled data size of S (defined in §III) because this size is robust in both theory [41, 42] and our evaluation on general queries (§VI-C); Our evaluation answers the following questions:

§VI-B: How accurate is UPA (in terms of RMSE) compared with FLEX?

§VI-C: What is the source of UPA's inaccuracy in estimating sensitivity?

§VI-D: What is UPA's performance overhead?

§VI-E: What are UPA's limitations?

B. UPA *v.s. FLEX*

We first compared the accuracy between UPA and FLEX. Figure 2(a) shows the RMSE of the local sensitivity values inferred by UPA compared with that of FLEX. UPA incurred on average 3.81% RMSE for all queries, which was five orders of magnitude lower than FLEX. Since the noise added to an output of a DP system is proportional to a sensitivity value, a small RMSE between UPA's inferred sensitivity value and the ground-truth sensitivity value (§II-A) indicates that UPA enables high utility for its query outputs. Note that the local sensitivity values inferred by UPA with its other components (§IV) enforce end-to-end iDP for queries.

For TPCH21, UPA's RMSE was six orders of magnitude smaller than FLEX's. UPA obtained much lower RMSE than FLEX because UPA inferred the sensitivity values based on a query's actual logic (§V), while FLEX inferred the sensitivity values statically based on the composition set of a query's operators, and ignored all control and data flow of a query, such as `Filter` and `Join` operator (§II-B). Thus, UPA inferred a more accurate sensitivity, while FLEX often tremendously overestimated the sensitivity value of queries which have multiple `Filter` operators and `Join` operators (such as TPCH16 and TPCH21). FLEX's paper [14] also confirmed that FLEX incurred high RMSE for TPCH16 and TPCH21 because these queries contain multiple `Filter` operators and `Join` operators.

Nonetheless, FLEX achieved zero error for TPCH1 because TPCH1 simply counts the number of tuples in a table (it has no `Filter` operator or `Join` operator), so FLEX directly returned one as TPCH1's local sensitivity value (i.e., a `Count` query's output at most changes by one when a data record is added to or removed from an input dataset). Overall, UPA is more general (supports all nine queries) and much more accurate than FLEX.

C. *Analysis of* UPA*'s accuracy*

To understand why the sensitivity values inferred by UPA had varied RMSE, we collected the output values of each of the nine queries on all the neighbouring datasets of each input dataset repetitively. Specifically, in this part of the evaluation, each input dataset had 200k records, so each input dataset had 200k neighbouring datasets and we collected 200k output values for each query, shown in Figure 3 (the output values of "KMeans" were not shown because their distribution was almost identical to "Linear Regression"). The spots in the figures were the output values. The red lines show the maximum and minimum output values of neighbouring dataset inferred by UPA with the default 1000 sample sizes n, and the blue lines show the ground truth greatest and smallest output values. Lines of other colours show the maximum and minimum output values of neighbouring datasets inferred by UPA with different sample sizes. Thus, if the red lines are closer to the blue lines, UPA infers a more accurate sensitivity value. Overall, for different sample sizes (10^2-10^5), when $n = 1000$, the red lines covered 98.9% $\sim$ 100.0% output values of all the neighbouring datasets except for TPCH21, indicating that $n = 1000$ is sufficient to infer an accurate sensitivity value. This size is also suggested by statistics theories [41, 42].

Among all queries, the output range inferred by UPA (red lines) for TPCH1 was the most accurate, because the actual output range of neighbouring dataset (blue lines) is small and the output values are uniformly distributed within the range. It is because TPCH1 is a simple count query; adding or removing one record from the dataset causes the output values at most change by one. Actually, UPA's sampling error of the output values of TPCH1 on neighbouring datasets was zero with sample size of 1000 (i.e., the output values sampled by UPA covered all the distinct values of all the output values of TPCH1's neighbouring dataset). Nevertheless, UPA incurred $2.6 * 10^{-09}$ RMSE (see Figure 2(a)) because UPA inferred the output range (red lines) by fitting a normal distribution to the sampled output values, while the output values of TPCH1 on neighbouring datasets may not perfectly follow a normal distribution (§IV-A).

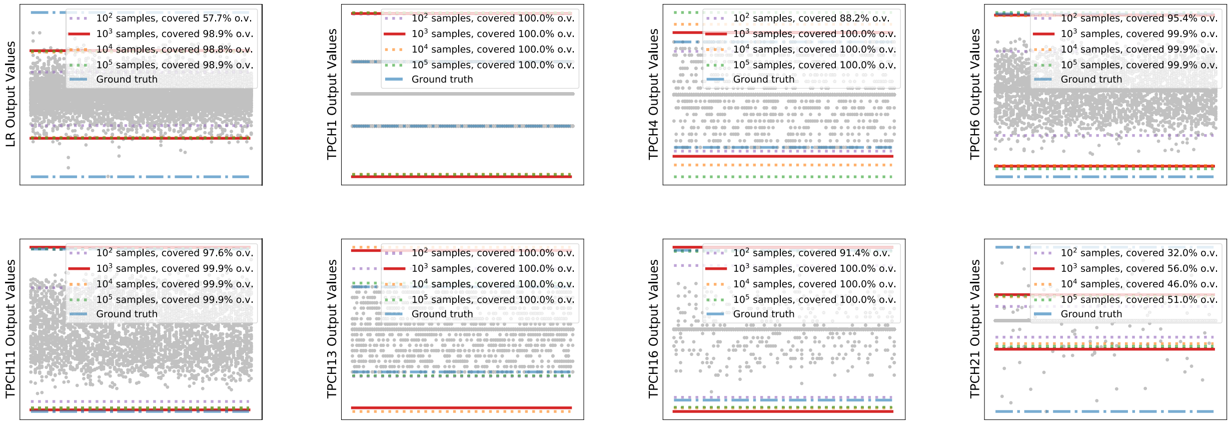

Fig. 3: Output values of queries with one record being added to or remove from the input dataset.

For all queries, the output range inferred by UPA for TPCH21 was the least accurate, because the neighbouring datasets' output values of TPCH21 have a wider range and the output values' distribution is non-uniform (there are outliers). It is because TPCH21 has 3 `Filter` operators and 5 `Join` operators, so the input data records have uneven effects on TPCH21's output value. These outliers have a low probability of being sampled by UPA, and are unlikely covered by the normal distribution inferred by UPA based on sampled output values (which are mostly not outliners). Nevertheless, RANGE ENFORCER enforces the output range to the query's output (§IV-B), and therefore UPA enforces iDP even the inferred sensitivity value is not close to ground truth. For the other eight queries, UPA enforces iDP with accurate sensitivity values.

D. *Performance Overhead of* UPA

We study the performance overhead of UPA in enforcing end-to-end DP. UPA's end-to-end computation time contains three parts: 1) computing a query's vanilla output, 2) computing the output values of sampled neighbouring datasets (§III) for adding noise, and 3) RANGE ENFORCER's action.

To analyse UPA's end-to-end performance overhead, we ran UPA on each query for 100 times and measured the execution time. For each execution, we randomly added or removed one to two data records in the input dataset, such that RANGE ENFORCER's two cases have equal probability to occur (§IV-B).

Figure 2(b) shows UPA's computation time normalized to vanilla Spark's on the default dataset size (Table V). UPA incurred 19.1% $\sim$ 130.9% performance overhead (on average 77.6%). In comparison, because FLEX is a static approach, it reported a 0.03% performance overhead [14], but it supports only five out of the nine evaluated queries and has orders of magnitude higher RMSE than UPA on inferring sensitivity values.

UPA incurred higher than 50% performance overhead for LR, KM, TPCH1, TPCH4, TPCH6 and TPCH13. For LR, KM, TPCH1 and TPCH6, all computations of these queries are local computation (no data record shuffling in Spark is triggered in these queries' logic, see §V). Since RANGE ENFORCER introduces additional data record shuffling in Spark: RANGE ENFORCER needs to exchange the data records which belong to the same partition between computers (§IV-B), the shuffling incurs substantial performance overhead compared to local computation. Hence, UPA's performance overhead on LR, KM, TPCH1 and TPCH6 was large. We observed for LR, KM, TPCH1 and TPCH6, more than 42.8% of the execution time spent in shuffling, which implies UPA's performance overhead on these queries mainly came from RANGE ENFORCER.

For TPCH4 and TPCH13, since they contain `Join` operators, they incurred more than 100% overhead. It is because UPA triggers shuffling (exchanging tuples between computers) twice (§V-C) for computing the joined results of sampled neighbouring datasets; vanilla Spark requires just one shuffle for one `Join`.

Surprisingly, TPCH16 and TPCH21, which are composed of more `Join` operators than TPCH4 and TPCH13, had lower performance overhead than TPCH4 and TPCH13. It is because most records (more than 99%) were filtered by consecutive `Filter` and `Join` operators (i.e., most immediate computation values of the sampled neighbouring datasets were filtered). Hence, UPA's overhead was small in computing the output values of sampled neighbouring datasets (line 10$\sim$14 of Algorithm 1).

We then study UPA's scalability to dataset sizes. Figure 4(a) shows UPA's performance overhead running on different dataset sizes. UPA had smaller performance overhead when dataset sizes were larger because the performance overhead of inferring sensitivity is constant (n is 1000). Even with a larger sample size, UPA's performance overhead did not increase significantly.

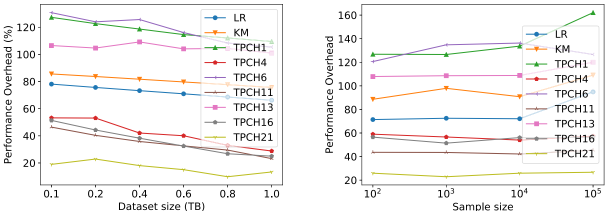

Fig. 4: UPA's performance scalability to dataset sizes (left) and sample size n (right).

Figure 4(b) shows UPA's performance remained constant for most queries up to a sample size of 10^5. It is because inferring sensitivity requires repeatedly computing the sampled data records, which results in a high hit rate of Spark's memory cache (cache hit rate increased from 10.3% to 48.9% when UPA entered line 10∼14 of Algorithm 1.

E. Limitations of UPA

Although UPA automatically and accurately infers a local sensitivity value and enforce DP to big data queries, UPA has two limitations.

First, UPA incurred moderate performance overhead. Although UPA's performance overhead was non-trivial, it is worthwhile and is already near-optimal. UPA's performance overhead is worthwhile because UPA addressed two main limitations of FLEX: UPA supports more general big data queries (Table II), and UPA infers a much more accurate sensitivity value (Figure 2a). It is because UPA is a dynamic analysis approach while FLEX is a static analysis approach.

Since dynamic analysis inevitably requires substantial additional computation [14, 15], UPA's performance overhead is also inevitable. However, UPA's performance overhead is already near-optimal. Specifically, occurring work [14, 15] pointed out that the performance overhead of accurately inferring a sensitivity value is at least linearly proportional to input datasets' size, and UPA leverages the associativity and commutativity properties of big-data operators, as well as sampling theories to reduce such performance overhead from linear to constant (i.e., UPA's performance overhead is independent of input datasets' size, as confirmed in Figure 4). It is because regardless of input datasets' size, UPA only needs to infer a local sensitivity value from 1000 sampled data records (§IV-A). Hence, UPA's performance overhead is considered near-optimal that UPA already reduces the performance overhead from linear to constant.

Second, UPA focuses on enforcing DP for an individual's data record, rather than a group of data records. Actually, other DP systems [32–34] also have the same focus. In future work, UPA can be extended to enforce DP for a group of individuals by reusing the results computed from the sampled neighbouring datasets across repeated queries.

VII. Related Work

Inferring sensitivity has been a challenge for DP since DP was introduced. Existing systems usually require expert involements to infer a sensitivity value. Airavat [19], GUPT [18], PINQ [46] enforce DP on data mining systems (e.g., MapReduce [36]). These systems require a data analyst, who submitted a query to these systems, to estimate an output range for the query, so that the systems can derive a global sensitivity value for the query (§IV-B).

Fuzz [47, 48] aims to ease manual effort in inferring a query's output range. It requires a data analyst to estimate the output range of a query's operators, so that Fuzz can derive an output range for the query based on the composition of the operators. However, this approach is not applicable to one-to-many and many-to-many operators (e.g., `Join`) [14]. It is because the influence of an input record on these operators' output value is usually unbounded. Tensorflow-Privacy [49, 50] enforces DP on stochastic gradient descent algorithms, but also requires a sensitivity value to be manually estimated. Compared to these approaches, UPA automatically infers a sensitivity value and enforces end-to-end DP.

Other DP systems (SensitivitySampler [15] and FLEX [14]) aim to reduce manual effort in inferring a sensitivity value. SensitivitySampler enforces Random Differential Privacy (RDP) [43] for general big-data queries. However, it has two major shortcomings. First, SensitivitySampler requires expert knowledge in statistic to identify a proper distribution for inferring a global sensitivity value. Second, SensitivitySampler provides DP guarantee to an input dataset probabilistically (i.e., enforces RDP). SensitivitySampler cannot practically be extended to enforce standard DP [5] and iDP [31], because RDP theoretically needs to add infinite noise to an output value to provide the standard DP and iDP guarantee. FLEX infers sensitivity values for a limited set of SQL queries (§II-B). Different from these two systems, UPA automatically enforces iDP for general big-data queries.

VIII. Conclusion

We have presented UPA, the first automated, accurate, and efficient DP big-data computation system. Evaluation on diverse big-data mining queries and comparison with relevant systems showed that UPA is deployable and easy to use. UPA has the potential to greatly promote the adoption of DP in real-world big-data analytics, effectively protecting the privacy of user data. All UPA's source code and evaluation results are available on https://github.com/hku-systems/UPA.

Acknowledgment

We thank anonymous reviewers for their helpful comments. Heming Cui is the corresponding author of this paper. This work is funded in part by one research grant from the Huawei Innovation Research Program (HIRP) Flagship, HK RGC ECS (27200916), HK RGC GRF (17207117 and 17202318), Croucher Innovation Award, National Natural Science Foundation of China (No. 61872318) and National Key R&D Program of China (No.2018YFB1004003).

References

[1] J. Jianyu, Z. Shixiong, A. Danish, W. Yuexuan, C. Heming, L. Feng, and G. Zhaoquan, "Kakute: A precise, unified information flow analysis system for big-data security," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC '17)*, 2017.

[2] X. C. Jianyu Jiang, C. W. Tzs On Li, T. Shen, S. Zhao, C.-L. W. Heming Cui, and F. Zhang, "Uranus: Simple, efficient sgx programming and its applications," in *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS '20)*, 2020.

[3] A. G. Martinez, *Chaos monkeys: Obscene fortune and random failure in Silicon Valley*. HarperCollins Publishers, 2018.

[4] A. Narayanan and V. Shmatikov, "How to break anonymity of the netflix prize dataset," *arXiv preprint cs/0610105*, 2006.

[5] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 265–284. [Online]. Available: http://dx.doi.org/10.1007/11681878_14

[6] R. Chen, Q. Xiao, Y. Zhang, and J. Xu, "Differentially private high-dimensional data publication via sampling-based inference," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 129–138.

[7] H. Li, L. Xiong, and X. Jiang, "Differentially private synthesization of multi-dimensional data using copula functions," in *Advances in database technology: proceedings. International Conference on Extending Database Technology*, vol. 2014. NIH Public Access, 2014, p. 475.

[8] W. Qardaji, W. Yang, and N. Li, "Priview: practical differentially private release of marginal contingency tables," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1435–1446.

[9] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao, "Privbayes: Private data release via bayesian networks," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, p. 25, 2017.

[10] C. Xu, J. Ren, Y. Zhang, Z. Qin, and K. Ren, "Dppro: Differentially private high-dimensional data release via random projection," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 3081–3093, 2017.

[11] P. Dandekar, N. Fawaz, and S. Ioannidis, "Privacy auctions for inner product disclosures," *CoRR, abs/1111.2885*, 2011.

[12] R. Kumar, R. Gopal, and R. Garfinkel, "Freedom of privacy: anonymous data collection with respondent-defined privacy protection," *INFORMS Journal on Computing*, vol. 22, no. 3, pp. 471–481, 2010.

[13] Z. Jorgensen, T. Yu, and G. Cormode, "Conservative or liberal? personalized differential privacy," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1023–1034.

[14] N. Johnson, J. P. Near, and D. Song, "Towards practical differential privacy for sql queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 526–539, 2018.

[15] B. I. Rubinstein and F. Alda, "Pain-free random differential privacy with sensitivity sampling," *arXiv preprint arXiv:1706.02562*, 2017.

[16] F. McSherry and I. Mironov, "Differentially private recommender systems: building privacy into the net," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 627–636.

[17] A. Yargic and A. Bilge, "Privacy risks for multi-criteria collaborative filtering systems," *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[18] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "Gupt: Privacy preserving data analysis made easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213876

[19] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for mapreduce," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855731

[20] J. Wang and Q. Tang, "Differentially private neighborhood-based recommender systems," *ICT Systems Security and Privacy Protection IFIP Advances in Information and Communication Technology*, p. 459–473, 2017.

[21] Y. Shen and H. Jin, "Epicrec: towards practical differentially private framework for personalized recommendation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 180–191.

[22] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3&#8211;4, pp. 211–407, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042

[23] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 111–125.

[24] U. Weinsberg, S. Bhagat, S. Ioannidis, and N. Taft, "Blurme," *Proceedings of the sixth ACM conference on Recommender systems - RecSys 12*, 2012.

[25] H. Polat and W. Du, "Privacy-preserving collaborative filtering using randomized perturbation techniques," *Third*

IEEE International Conference on Data Mining.

[26] ——, "Privacy-preserving top-n recommendation on horizontally partitioned data," *The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI05)*.

[27] ——, "Achieving private recommendations using randomized response techniques," *Advances in Knowledge Discovery and Data Mining Lecture Notes in Computer Science*, p. 637–646, 2006.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[29] "Hadoop," http://hadoop.apache.org/core/.

[30] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," *Proc. LSDS-IR*, vol. 8, 2009.

[31] J. Soria-Comas, J. Domingo-Ferrer, D. Sánchez, and D. Megías, "Individual differential privacy: A utility-preserving formulation of differential privacy guarantees," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1418–1429, 2017.

[32] H. H. Nguyen, J. Kim, and Y. Kim, "Differential privacy in practice," *Journal of Computing Science and Engineering*, vol. 7, no. 3, pp. 177–186, 2013.

[33] A. Machanavajjhala, X. He, and M. Hay, "Differential privacy in the wild: A tutorial on current practices & open challenges," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1727–1730.

[34] C. Dwork and J. Lei, "Differential privacy and robust statistics," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM, 2009, pp. 371–380.

[35] K. Nissim, S. Raskhodnikova, and A. Smith, "Smooth sensitivity and sampling in private data analysis," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. ACM, 2007, pp. 75–84.

[36] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 10–10.

[37] Z. Xu, M. Hirzel, and G. Rothermel, "Semantic characterization of mapreduce workloads," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 87–97.

[38] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 29–42.

[39] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[40] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2018.

[41] S. L. Braunstein, "How large a sample is needed for the maximum likelihood estimator to be approximately gaussian?" *Journal of Physics A: Mathematical and General*, vol. 25, no. 13, p. 3813, 1992.

[42] R. A. Hart and D. H. Clark, "Does size matter? exploring the small sample properties of maximum likelihood estimation," in *Annual Meeting of the Midwest Political Science Association*. Citeseer, 1999, pp. 1–32.

[43] R. Hall, A. Rinaldo, and L. Wasserman, "Random differential privacy," *arXiv preprint arXiv:1112.2680*, 2011.

[44] R. Newson, "Confidence intervals for rank statistics: Percentile slopes, differences, and ratios," *The Stata Journal*, vol. 6, no. 4, pp. 497–520, 2006.

[45] K. M. Robinette and J. T. McConville, "An alternative to percentile models," *SAE Transactions*, pp. 938–946, 1981.

[46] F. McSherry, "Privacy integrated queries," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, Inc., June 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/privacy-integrated-queries/

[47] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 33–33. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028100

[48] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear dependent types for differential privacy," in *Acm sigplan notices*, vol. 48, no. 1. ACM, 2013, pp. 357–370.

[49] "Tensorflow Privacy," https://github.com/tensorflow/privacy.

[50] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 308–318. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978318